

USENIX

MACH III SYMPOSIUM PROCEEDINGS

U

US

USE

U

NIX

USENIX

## SYMPOSIUM PROCEEDINGS

Mach III

April 19-21, 1993  
Santa Fe, New Mexico

SPRING  
1993

For additional copies of these proceedings contact:

USENIX Association  
2560 Ninth Street, Suite 215  
Berkeley, CA 94710 U.S.A.

The price is \$30 for members and \$39 for non-members.

Outside the U.S.A and Canada, please add  
\$18 per copy for postage (via air printed matter).

Past USENIX Mach Proceedings (price: member/nonmember)

Mach Workshop	October 1990	Burlington, VT	\$17/20
Mach II	November 1991	Monterey, CA	\$24/28

Outside the U.S.A. and Canada, please add  
\$9 per copy for postage (via air printed matter).

Copyright © 1993 by The USENIX Association  
All rights reserved.

ISBN 1-880446-49-9

This volume is published as a collective work.

UNIX is a registered trademark of UNIX System Laboratories, Inc.

USENIX acknowledges all trademarks appearing herein.

Printed in the United States of America on 50% recycled paper, 10-15% post-consumer waste.





# **Proceedings of the USENIX Mach III Symposium**

**USENIX Association**

**April 19-21, 1993  
Santa Fe, New Mexico, U.S.A.**



# MACH III SYMPOSIUM

April 19-21, 1993

Santa Fe, New Mexico

## Table of Contents

### Opening Remarks

*David L. Black, Open Software Foundation*

### Keynote Address: Operating Systems for Operating Systems Research

*Sape J. Mullender, University of Twente, Netherlands*

### Experiments with Real-Time Servers in Real-Time Mach.....1

*Tatsuo Nakajima, Takuro Kitayama, and Hideyuki Tokuda, Carnegie Mellon University*

### UNIX File Access and Caching in a Multicomputer Environment.....21

*Paul J. Roy, Open Software Foundation*

### In-Kernel Servers on Mach 3.0: Implementation and Performance.....39

*Jay Lepreau, Mike Hibler, Bryan Ford, and Jeffrey Law, University of Utah*

### Redirecting System Calls in Mach 3.0, An Alternative to the Emulator.....57

*Simon Patience, Open Software Foundation*

### A Fast and General Implementation of Mach IPC in a Network.....75

*Hilarie Orman, Edwin Menze III, Sean O'Malley, Larry Peterson, University of Arizona*

### Port Buffers: A Mach IPC Optimization for Handling Large Volumes of Small Messages.....89

*Kenneth W. Koontz, The Johns Hopkins University*

### Using the Mach Communication Primitives in X11.....103

*Michael Ginsberg, Robert V. Baron, and Brian N. Bershad, Carnegie Mellon University*

### Real Time - Mach Timers: Exporting Time to the User.....111

*Stefan Savage and Hideyuki Tokuda, Carnegie Mellon University*

### Adding Scheduler Activations to Mach 3.0.....119

*Paul Barton-Davis, Dylan McNamee, Raj Vaswani, and Edward D. Lazowska,  
University of Washington*

### Using Continuations to Build a User-Level Threads Library.....137

*Randall W. Dean, Carnegie Mellon University*

### An Architecture for Device Drivers Executing as User-Level Tasks.....153

*David B. Golub, Carnegie Mellon University; Guy G. Sotomayor, Jr, and  
Freeman L. Rawson III, IBM*

MVM-An Environment for Running Multiple DOS, Windows and DPMI Programs on the Microkernel.....	173
<i>David Golub, Carnegie Mellon University; Ravi Manikundalam and Freeman Rawson III, IBM</i>	
An OS/2 Personality on Mach.....	191
<i>James M. Phelan, James Arendt, and Gary R. Ormsby, IBM</i>	
Page Prefetching Based on Fault History.....	203
<i>Inshik Song and Yookun Cho, Seoul National University, Korea</i>	
Kernel Support for Recoverable-Persistent Virtual Memory.....	215
<i>Khien-Mien Chew, University of Texas - Austin, Jyothy Reddy, Hewlett-Packard; Theodore H. Romer and Abraham Silberschatz, University of Texas - Austin</i>	
Real Memory Mach.....	235
<i>Philippe Bernadat and David L. Black, Open Software Foundation</i>	
MIKE: A Distributed Object-Oriented Programming Platform on Top of the Mach Microkernel.....	253
<i>Miguel Castro, Nuno Neves, Pedro Trancoso, and Pedro Sousa, INESC, Portugal</i>	
Task Migration on the Top of the Mach Microkernel.....	273
<i>Dejan Milojicic, Wolfgang Zint, Andreas Dangel, and Peter Giese, University of Kaiserlautern, Germany</i>	
The Design of the Schizophrenic Workstation System.....	291
<i>Mark Swanson, Leigh Stoller, Terence Critchlow, and Robert Kessler, University of Utah</i>	
Sprite on Mach.....	307
<i>Michael D. Kupfer, University of California at Berkeley</i>	

### **Program Committee**

David L. Black, Chair, Open Software Foundation  
David B. Golub, Carnegie Mellon University  
Alan Langerman, Orca Systems, Inc.  
Jay Lepreau, University of Utah  
Avadis Tevanian, Jr., NeXT, Inc.

# Experiments with Real-Time Servers in Real-Time Mach

Tatsuo Nakajima, Takuro Kitayama and Hideyuki Tokuda  
*School of Computer Science  
Carnegie Mellon University  
Pittsburgh, Pennsylvania 15213*

## Abstract

The Mach micro kernel allows many operating system functions such as file systems, network protocols, TTY managers and process managers to be implemented as user level servers. Application programmers can write their own operating system servers suitable for their own applications. Mach, however, does not provide the mechanisms for building the servers which ensure predictable services. Therefore, if real-time applications which have rigid timing constraints access the servers, they will fail to meet deadlines.

Real-Time Mach provides a set of mechanisms and policies for implementing real-time servers which provide predictable services. In this paper, we present a model for real-time servers for providing predictable services, system supports for implementing the model, and the evaluation of the implementation. We also describe the experiments with building operating system servers based on the model in Real-Time Mach.

## 1 Introduction

The micro kernel based operating system architecture is widely recognized as a promising approach for providing better modularity and extensibility. A micro kernel provides basic resource management functions such as processor scheduling, memory object management, IPC facilities, and low-level I/O support. Traditional functions of operating systems such as file services and network services are all implemented in server tasks which run as user-level application programs[2].

Traditional operating system technologies focus on providing virtually infinite resources to users. The technologies are based on round-robin policy for CPU scheduling, FIFO queueing for managing blocked activities and message queueing, and LRU policy for memory replacement[17]. The technologies assume that the total working sets of tasks do not exceed the amount of physical resources. If the total resources required by applications exceed the amount of physical resources, the behavior of a system becomes unstable and the response time becomes tremendously slower. Current trends for extending operating system technologies toward multimedia applications require much bigger bandwidth and rigid timing constraints. Under traditional technologies, we cannot predict and analyze the timing behaviors of the applications. Future operating systems need to control the traffic generated by such high bandwidth applications and sophisticated technologies for controlling physical resources such as prioritized resource

---

<sup>1</sup> This research was supported in part by the U.S. NRaD under contract number N66001-87-C-0155, by the Office of Naval Research under contract number N00014-84-K-0734, by the Defense Advanced Research Projects Agency, ARPA Order No. 7330 under contract number MDA72-90-C-0035, and by the Federal Systems Division of IBM Corporation under University Agreement YA-278067. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing official policies, either expressed or implied, of NRaD, ONR, DARPA, IBM, Northrop, Bellcore, or the U.S. Government.

scheduling and admission controls[3] will become more important. We believe that incorporating real-time technologies into general operating systems is a first step toward the goal.

The advantage of using a micro kernel for real-time applications is that the preemptability of the kernel is better, the size of the kernel becomes much smaller, and the addition and the removal of servers is easier[1]. An embedded real-time system might remove a file system module if it is not used in applications where as multimedia applications might require full Unix functionalities.

However, such a micro kernel alone cannot provide a predictable real-time computing environment due to many unpredictable delays caused by unbounded priority inversion[20]. Traditional real-time operating systems support fixed priority scheduling, fast interrupt handling, and preemptable kernels. These facilities are important for building predictable real-time systems, but predictability and analyzability cannot be achieved by them alone.

Real-Time Mach[20, 21] provides a real-time thread package for periodic activities and ITDS scheduler for ensuring its timing constraints. They make it very easy to write applications with timing constraints. Mach provides MIG and C-Threads for building operating system servers[12], but they do not enable us to write predictable servers, since the scheduling policy and the locking protocol do not support real-time applications. In the servers using the Mach facilities, the highest priority request must wait for the completion of all existing requests, since incoming requests are enqueued in FCFS order. It makes very difficult to bound the worst case blocking time. Real-time applications need to access operating system servers for accessing files, creating new processes and communicating with other machines. If they are not provided, application programmers must write them from a scratch for themselves.

In this paper, we propose a real-time server model and system support for implementing the model in Real-Time Mach. We focus only on the structuring aspect of real-time servers. Other aspects such as the detailed algorithms of the priority management are discussed in another paper[11].

We also discuss the experiences with building two servers based on the real-time server model in Real-Time Mach. The first server is Real-Time Server(RTS) which supports a process manager, a memory based file system, a device manager, and a simple command interpreter. The second server is the Network Protocol Server(NPS) which implements network protocols such as UDP/IP. We created two versions of these servers; one using C-Threads and the original Mach IPC and the other using the real-time thread package and Real-Time IPC, to compare real-time servers with traditional servers.

The remainder of this paper is structured as follows. In section 2, we present the overview of kernel extensions and computational environments for Real-Time Mach. In section 3, we identify the problems in traditional servers and propose a real-time server model. Section 4 describes the implementation of the system supports for real-time servers. We also show the structures of RTS and NPS. In section 5, we describe the basic costs of the kernel primitives and the performance of RTS and NPS. Section 6 presents related work, and we summarize the paper in section 7.

## 2 Real-Time Mach Overview

Real-Time Mach[20, 21] provides functionalities for distributed real-time computing. In this section, we describe an overview of these functionalities.

### 2.1 Kernel Extensions

Real-Time Mach kernel has been developed at CMU to provide a common distributed real-time computing environment. Real-Time Mach is an extension of the Mach kernel and has the following five characteristics over the original Mach kernel.

- Real-Time Thread Model.
- Real-Time Scheduling.
- Clock and Timer.

- Real-Time Synchronization.
- Real-Time IPC.

The major feature of Real-Time Mach is predictable resource management which enables us to analyze behavior before executing applications. The resources for time critical threads are eager-evaluated. Every object provided by the kernel such as a thread, a memory and a port has attributes which reflect the requirements of applications.

In Real-Time Mach, a thread is defined for a real-time or non-real-time activity. For a real-time thread, additional *timing attributes* must be defined by a timing attribute descriptor. A real-time thread is classified as a *periodic* or an *aperiodic* thread, and each class of threads is defined as a *soft* or *hard* real-time.

The real-time scheduler in Real-Time Mach allows a system designer to predict whether the given task set can meet its deadlines or not. For soft real-time activities, the designer may predict whether the worst case response times meet the timing requirements or not. We adopted a *capacity preservation* scheme to cope with both hard and soft real-time activities. By capacity preservation, we mean that we divide the necessary processor cycles between the two types. Under a transient overload condition, the scheduler uses threads' *importance value* to decide which thread should complete its computation and which should be aborted or canceled.

A clock provides the abstraction of physical clocks which measure the passage of time. Clock allows the kernel to export high resolution timing hardware to users, which are important for the measurement of performance, the calculation of CPU usage, and fine-grained timestamping. A timer provides a synchronization action when its expiration time is reached.

Traditional synchronization primitives use FIFO ordering for queueing threads which wait for entering a critical region since FIFO ordering can avoid starvation. In a real-time computing environment, however, FIFO ordering often causes a priority inversion problem[16]. Real-time operating systems should support various synchronization policies based on the queueing order for waiting threads and the preemptability of running threads in a critical region.

IPC is heavily used in a micro kernel environment. A predictable IPC is important for building modular and manageable systems. Real-Time IPC can control the receiver of a message and the priorities between programs. When creating a new port, we can specify a port attribute to select IPC policies.

More detailed information is included in [20, 21, 14].

## 2.2 Computational Environment

Real-Time Mach provides two computational environments for real-time applications(Figure 1). The first computational environment is Unix. Unix environment enables application programmers to compile, test, and debug using Unix tools such as cc, make and gdb without rebooting machines. The environment is appropriate for debugging logical aspects of programs and makes the speed of developing programs very fast. However, the Unix server in Mach (UX server) is written using C-Threads and Mach IPC. Because they do not provide bounded response time, applications cannot use Unix functionalities such as files and sockets for the actual execution.

Real-time applications need high level functionalities such as files, TTY, processes, networks and command interpreters. We provide RTS and NPS for the purpose. RTS is a run-time environment for real-time applications on Real-Time Mach. Since RTS is implemented as a real-time server, the response time can be bounded. RTS offers process management, file management and TTY management like Unix, but it has several characteristics different from the Unix environment. RTS includes a simple Unix like command interpreter which runs as a thread in RTS. It saves physical resources and makes the command interpreter to run without file systems supporting persistent files. File systems in RTS provide a memory based file system and a read-only file system for reading Unix files from disks.

Real-Time applications may require a more powerful environment than the basic RTS environment. RTS provides the mechanism for extending the basic functionalities. Its application interface enables us to create a more sophisticated command interpreter which is running as a separate process. RTS also

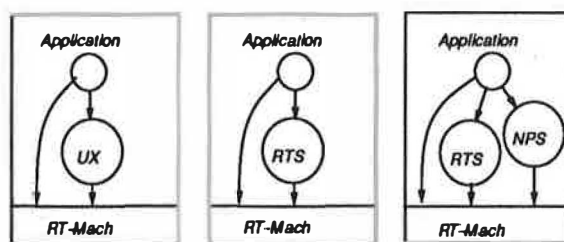


Figure 1: RT-Mach Environment

provides external file interface which can mount file servers running as separate processes. For example, UFS server which is a user level implementation of the Unix file system provides the read/write access of Unix files on disks.

NPS is a network server which is running on RTS. NPS enables applications to communicate with other applications on different machines. NPS consists of several layers which are separated by clean interfaces. It makes it easy to incorporate and experiment with a new protocol, even though the current version of NPS supports only UDP/IP protocol. Since NPS is also implemented as a real-time server, requesting to NPS does not cause unbounded priority inversion.

Because RTS and NPS are implemented as different servers, applications can select whether they use only RTS or both RTS and NPS. The flexibility provided by RTS and NPS environment is suitable for satisfying the various requirements of real-time applications.

### 3 Real-Time Server Model

In this section, we describe Mach server model and present several problems of the model in real-time environment. Then, we propose a real-time server model and its system support.

#### 3.1 Priority Inversion

A real-time system designer needs to determine the worst case blocking time of a higher priority activity for shared resources such as servers. It is often impossible to compute the bound if an activity in the protected region is preemptable. For realizing predictable and analyzable computing, blocking time should be bounded. Real-time systems control activities according to their deadlines. The unbounded blocking is caused by the preemption of lower priority activities which is acquiring a critical region. The problem is called *priority inversion problem*[16]. A similar situation is caused when clients send RPC messages to a server.

Let us consider the following case. Suppose that the lowest priority thread  $T_L$  sends a request to server  $S$  at  $t_1$ , then the highest priority thread  $T_H$  becomes runnable at  $t_2$  and attempts to send a request to server  $S$  at  $t_3$ . However, since the request of  $T_L$  is executed in the server,  $T_H$  must wait for its completion. After  $T_L$  resumes, a medium priority thread  $T_M$  becomes runnable at  $t_4$ . Then  $T_M$  starts running without accessing the server and may wake up another medium priority thread and so on. After  $T_M$  is completed at  $t_5$  and then server  $S$  returns a reply to  $T_L$  at  $t_6$ , the request of  $T_H$  starts to be processed. In the example, priority inversion is occurred from  $t_3$  to  $t_6$ . The time of priority inversion



cannot be bounded without knowing all behaviors of related medium priority threads. Figure 2 shows the time sequence in the execution.

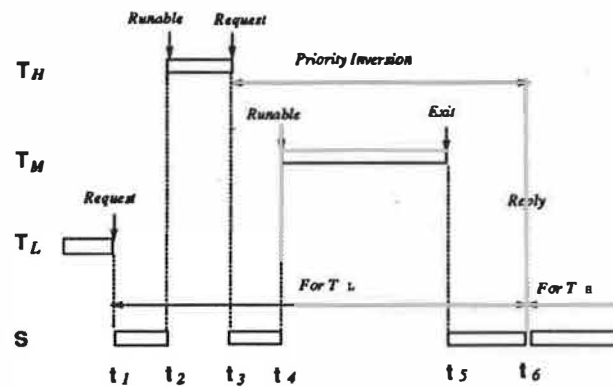


Figure 2: Priority Inversion in Servers (1)

In order to bound the worst case blocking time, the *priority inheritance* scheme was developed[16]. The priority inheritance scheme is that once  $T_H$  sends a request to server  $S$ , the server inherits the priority of  $T_H$ . Then,  $T_M$  cannot preempt the execution of the request of  $T_L$  in the server. In this way, the worst case blocking time of  $T_H$  can be bounded if only the worst case execution time for processing a request is known. Figure 3 shows the time sequence of the execution.

In traditional operating systems, FCFS policy is adopted for IPC message handling since fairness is important for providing virtually infinite resources to users and avoiding starvation. In real-time systems, however, FIFO ordering often creates the priority inversion problem. A higher priority activity must wait for the completion of all low priority activities in a waiting queue. If all real-time activities can meet their deadlines, there will be no starvation among these threads. Thus, the systems should provide a priority based ordering for queuing requests to servers to avoid the priority inversion problem.

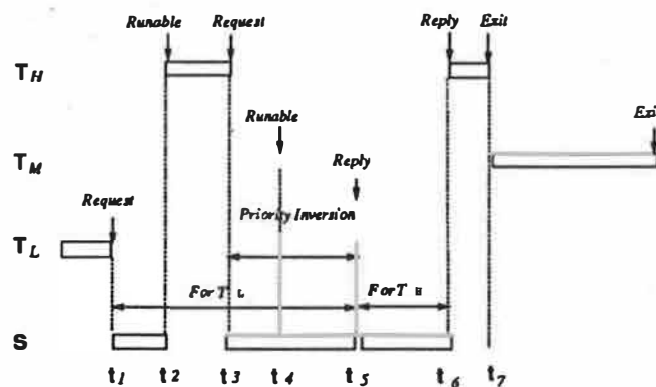


Figure 3: Priority Inversion in Servers (2)

### 3.2 Priority Inversion in Mach Servers

Mach implements operating system servers using C-Threads and MIG. C-Threads is a thread package implementing light-weight user level threads on top of kernel threads. MIG is a RPC stub generator for Mach IPC. Typical servers such as the CMU Unix server (UX server) create several threads for receiving requests from clients. When a thread processing a request is suspended in the middle of the execution, a new thread which processes another request may be created. A thread may wire itself for binding a user level thread to a kernel thread.

The structure of servers is appropriate for time sharing applications, but may cause unbounded priority inversion which makes the behavior of systems unpredictable and unanalyzable.

There are three major problems in the traditional server structure in Mach. The first problem is caused by IPC. When the priority of a server is lower than the priority of a client, the execution of a request may be preempted by a medium priority thread. Also, if a high priority request arrives at a server while the server is processing a request of a low priority client, a medium priority thread can preempt the execution of the server. These situations may cause unbounded priority inversion. The second problem is caused by the C-Threads package. C-Threads implements user level threads on kernel threads. This means that several C-threads are mapped onto one kernel thread. User level threads do not have the notion of a priority and they are scheduled by non-preemptable FIFO scheduling. This strategy does not satisfy the requirements of real-time applications. Moreover, selecting the next runnable highest priority thread in a system is difficult since user level schedulers do not know about the blocking of threads in the kernel. The third problem is caused by the lack of the coordination between synchronization and IPC. In Mach, synchronization is implemented in a user space and IPC is implemented in a kernel space. Because the blocking in IPC and critical regions are managed as independent events, it is difficult to implement the priority inheritance protocol which is integrated with IPC and synchronization in the environment. In Real-Time Mach, the synchronization is implemented in a kernel space, thus the synchronization, the IPC, and the scheduler can be closely integrated.

In order to solve these problems, we propose a real-time server model and system support to implement the model.

### 3.3 Real-Time Server Model

The real-time server model provides a framework for ensuring predictable service time. In this paper, we assume clients and servers are in the same machine and the machine has only one CPU. The model supports only synchronous communications because introducing asynchronous communications makes schedulability analysis difficult. The structure of servers and the priority management are key points in the model. We classify the model based on the structure of servers and describe the priority management in each class.

We classify the model into three classes. The classification is made by the number of threads in a server and the time to create a thread for processing requests.

The first class of the model is a *single thread server model*. In the model, one thread processes all incoming requests as showed in Figure 4. A new request cannot preempt the execution of a previous request executed in the server, then it is blocked until the termination of the previous request. A request is assigned the same priority as a sender thread, and the priority of the thread in the server is dynamically changed based on the priority of the request when receiving the request. If the thread is not waiting for a new request, the new request is enqueued in a prioritized queue and if the priority of the request is higher than the priority of the thread, the thread priority is raised to the priority level of the request. We assume that the priority level in a IPC domain is the same as the priority level in a thread scheduling domain, thus we can omit the problems caused by mapping priorities in different domains in this paper<sup>1</sup>. The model is easy to implement, but the length of priority inversion will become long if the execution times of requests are long.

<sup>1</sup>When a message is delivered to a remote machine, whether the assumption is ensured or not depends on the priority level supported by a network. If the priority level in a network domain is smaller than the priority level in a CPU scheduling

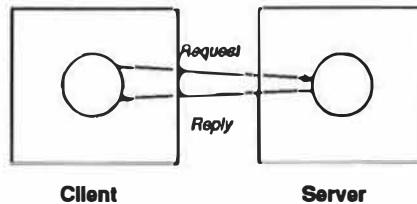


Figure 4: Single Threaded Server Model

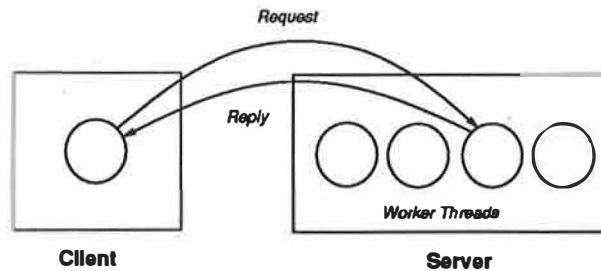


Figure 5: Worker Model

In the second class of model, several threads process requests as shown in Figure 5. We call the model a *worker model*. The threads which process requests are called *worker threads*. The class is further divided into two types.

The first type is a *static prioritized worker model*. In the model, each thread in a server is assigned a different priority. When a new request comes to the server, the server selects a thread which has the same priority as the priority of the request. If the thread is running, the request is blocked and the request is enqueued in a priority queue until the thread completes a previous request. The model provides better preemptability than the single threaded server model.

The second type is a *dynamic prioritized worker*. In the model, a thread receiving a request inherits the priority of a request. If all threads are processing requests, a new request must wait for the termination of one of the thread executing requests. A thread is selected to process the new request and the request is enqueued in a priority queue waiting for the completion of the selected thread. If the priority of the request is higher than the current priorities of the selected thread, the priority of the thread is bumped up to the priority of the new request. The problem of the model is that priority inversion occurs if all

domain, the mapping priorities between two domains are required. The effect of the mapping is reported in [4].

workers are used by low priority requests.

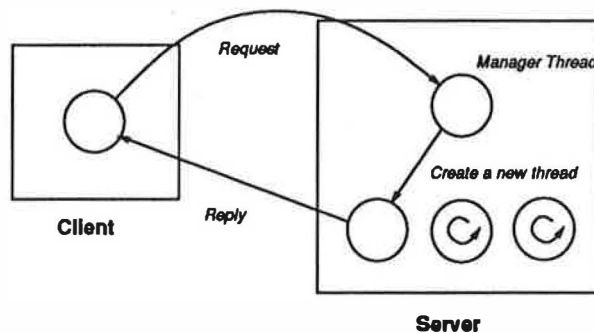


Figure 6: Dynamic Server Model

The last class of the model is a *dynamic server model*(Figure 6). In the model, a new thread is created whenever a new request is arrived. The newly created thread inherits the priority of the request. The model provides the best preemptability because a request does not need to wait for the completion of threads in servers unlike other models. However, the cost of the creation of a new thread is expensive if the thread is implemented as a kernel supported thread. Since usual IPC does not support the creation of new threads when a request comes, a manager thread is required for the creation of threads, which requires extra context switch. The most serious problem of the model is that a thread is created dynamically, then it is difficult to estimate the maximum resource usage in a server. This model is not suitable for real-time servers, so we use only the single threaded server model and the worker model in real-time servers.

The selection of the models is determined by the tradeoff between preemptability and the overhead caused by preemption. The single threaded server model is suitable for the server which has only short services. In the dynamic prioritized worker model, a high priority request may be blocked and needs to wait for the completion of a low priority request if all threads are used by low priority requests. On the other hand, in the static worker model, a thread for processing a high priority request is not dispatched for a low priority request so that the high priority request can preempt the low priority request at any time. However, if the number of worker threads is smaller than the number of priority levels of threads in a task set, each worker thread may need to execute requests which have different priorities. The situation may cause unbounded priority inversion because a high priority request may wait for a low priority request without inheriting the priority. The two models have different characteristics of priority inversion: priority inversion due to blocking and priority inversion due to mapping priority levels. Application programmers should select a suitable model by the characteristics of their applications. In [8], they discuss the same issues in the network protocol processing and show the detailed simulation results.

### 3.4 Schedulability Analysis

One of our goal is to provide a better interface to adopt well-known schedulability analysis techniques. For instance, given a set of periodic, independent tasks in a single processor environment, with the rate monotonic scheduling algorithm the worst case schedulable bound is 69%[6], the average case is 88% [5], and the best case, where threads have harmonic periods, is up to 100% of the CPU utilization. This means that the CPU utilization of all periodic tasks is under the schedulable bound, the deadlines of all the tasks will not be missed.

In the case of a more general task set where threads can synchronize via critical regions, we can also bound the synchronization (blocking) time for each task by using the priority inheritance protocol. Using

these inheritance protocols, we can also check the schedulable bound for  $n$  periodic threads as follows.

$$\forall i, 1 \leq i \leq n, \frac{B_i}{T_i} + \sum_{j=1}^i \frac{C_j}{T_j} \leq i(2^{\frac{1}{i}} - 1)$$

where  $C_i$ ,  $T_i$ ,  $B_i$  represents the total computation time, the period, and the worst case blocking time of *Thread<sub>i</sub>* respectively[16].

In the real-time server model, we can also use the above schedulability analysis formula because the blocking in servers is very similar to the blocking in synchronization. If we use the single threaded server model, the formula becomes identical. In the case of the static prioritized worker model, if the number of worker threads is equal to the number of priority levels, which is used in an application and all threads have different priorities, the first term for blocking in the above formula disappears. In either case, the formula may be more pessimistic than the actual worst cases.

### 3.5 System Supports for Real-Time Servers

Real-Time Mach provides system level support for implementing the real-time server model. The model requires three different policies in IPC, *dispatch policy*, *queueing policy* and *handoff policy*. The *dispatch policy* determines which thread in a server process a request. The policy is important to implement the worker model. The *queueing policy* determines the policy for the order of messages in queues. The policy also decides whether the priority of blocked request should be inherited to a thread in a server. The thread which inherits the priority of a request is determined by the dispatch policy. The *handoff policy* decides whether the priority of a request is inherited by a thread in a server or not. The combination of the above policies enables us to implement various real-time server models.

Another important aspect of system support is the integration of synchronization and IPC. In the worker model, blocking may occur in both IPC and critical regions. If the thread which is chosen by the dispatch policy is blocked to wait before entering a critical region, the priority of the request should be inherited by the thread which executes the critical region. The above policies and the integration of IPC and synchronization are implemented by a real-time resource manager. The Real-Time IPC module and the synchronization module implement only mechanisms and they call the real-time resource manager for executing policies for controlling priorities of threads. We discuss the implementations in the next section.

## 4 Implementation

In this section, we describe the system support for the real-time server model. The current version of the system support is implemented in kernel space. It is divided into the real-time resource manager module, the real-time IPC(RT-IPC) module and the real-time synchronization(RT-Synchronization) module. We also describe the structure of RTS and NPS.

### 4.1 Kernel Extensions

#### 4.1.1 Real-Time Resource Manager

Real-time resource manager implements a set of policies for maintaining the consistency of priority management in IPC and synchronization.

The basic idea of the real-time resource manager is to adopt the same abstraction and same functions in both RT-IPC and RT-synchronization. The abstraction in our model is a resource and a client. A client is the abstraction of the user of a resource. A resource represents logical resources such as a critical region or threads in a server. A resource is manipulated by the two functions: acquire and release. A client must call acquire before using it and release should be called after the use. When a client tries to acquire the resource which has been already acquired by another client, the execution of the client

is suspended until the resource is released. The suspension of the client may cause unbounded priority inversion. The real-time resource manager enables us to control the priorities of the client which is the owner of a resource according to the policy of the resource.

The real-time resource manager is divided into three components which correspond to the policies required for implementing real-time servers: resource dispatch module, queueing module, and priority handoff module. These three modules implement the policies described in section 3.5.

In the current version, the resource dispatch module supports *FIFO* policy and *WORKER* policy. The *FIFO* policy dispatches a free resource if it is available. If not, an arbitrary resource is chosen. The policy is used to implement the dynamic prioritized server model. The *WORKER* policy is used when implementing the static prioritized worker model. The resource is chosen by the priority of a client.

The queueing module provides *FIFO* policy, *PRIO* policy and *BPI* policy. The *FIFO* policy enqueues a client in a *FIFO* order. The *PRIO* policy and *BPI* policy enqueue a client in a priority order. The difference is that the *BPI* policy executes the priority inheritance protocol if a client is enqueued. The static prioritized server model uses *PRIO* policy and the *BPI* policy is adopted in the dynamic prioritized server model and the single threaded server model.

The priority handoff policy supports *HANDOFF ON* policy, *HANDOFF OFF* policy and *HANDOFF MSG*. The *HANDOFF ON* policy inherits the priority of one client to the priority of another client. The *HANDOFF OFF* policy does nothing. The *HANDOFF MSG* inherits the priority specified by programmers to another client. The policy is used in NPS for inheriting a priority included in a IP packet to a worker thread. The *HANDOFF ON* policy is used in the dynamic prioritized server model and the single threaded server model and the *HANDOFF OFF* policy is used in the static prioritized server model.

#### 4.1.2 Real-Time IPC

Real-Time IPC, or RT-IPC, is a modified version of Mach IPC. The real-time server model uses only synchronous communication through RT-IPC provides both asynchronous communication and synchronous communication.

In RT-IPC, the destination of IPC is represented by a port which is allocated by *rt\_mach\_port\_allocate*. A port attribute enables us to specify the policies to be used in the real-time resource manager, the number of buffers, and its size for blocked requests. *Rt\_mach\_msg* is called when threads send and receive messages. The arguments of the function are the same as the arguments of *mach\_msg*. A port must be associated with the thread which processes requests arriving to the port using *rt\_mach\_port\_associate* before receiving requests. Ports are grouped in a port set and a client can specify a port which belongs to the port set for sending a request. *Rt\_mach\_port\_associate* can also specify the priority which is used for implementing the static prioritized worker model. We need this function in order not to change the argument of *rt\_mach\_msg*.

In RT-IPC, we regard a thread which processes a request as a resource and a request message as a client. Before calling the function, a server needs to allocate resources in the real-time resource manager, where the number of resources is the same as the number of threads in the server. When a new request arrives at a port, one of the resources is assigned for processing the request by the resource dispatch module. RT-IPC module wakes up the thread associated with the returned resource for processing the request. If the resource has been acquired by another request, the request will be blocked and enqueued in a waiting queue. RT-IPC calls the queueing module for queueing the request. The queueing module executes the priority inheritance protocol when the policy specified in a port attribute. The resource is released for other requests after the termination of processing the request. It may wake up a blocked request. The priority handoff module is called in order to decide whether the priority of a request is inherited or not.

#### 4.1.3 Real-Time Synchronization

Real-Time Mach provides condition variables and mutex variables for synchronization. Because the current condition variables do not support real-time features, we discuss only mutex variables.

The mutex variables are implemented as kernel primitives in Real-Time Mach, because implementing the priority inheritance protocol requires close integration with the processor scheduler and a different scheduling policy requires a different priority compare function.

A critical region is represented as a mutex variable in Real-Time Mach. Each critical region needs to allocate a mutex variable before entering the critical region. At the beginning of the critical region, we must call *rt\_mutex\_lock* and *rt\_mutex\_unlock* should be called at the end of the critical region. Programmers can select a policy for their critical regions by mutex attributes. Each critical region allocates a resource in the real-time resource manager when *rt\_mutex\_allocate* is called and deallocates the resource by *rt\_mutex\_deallocate*. A client of a resource is the thread which enters a critical region. When *rt\_mutex\_lock* is executed, a resource representing a mutex variable is acquired, and the resource is released in *rt\_mutex\_unlock*. The queueing policy of blocked threads is determined by the queueing module. Because a mutex variable requires only one resource, the dispatch policy is not important. The current implementation of the synchronization module uses the FIFO policy as a default policy. Users need not to specify the policy in a mutex attribute. In RT-synchronization module, the priority handoff policy is not used.

## 4.2 Examples of Real-Time Servers

In this section, we describe the structure of RTS and NPS on Real-Time Mach. The two servers use different real-time server models and extend the basic models for several practical reasons. The servers use a modified version of MIG for generating stub codes for RT-IPC. In the section 5.2, we show the cost of the real-time server model by comparing a version which uses the real-time thread package and RT-IPC on RT-Mach with a version using the C-Thread package and the original Mach IPC on original Mach.

### 4.2.1 Structure of RTS

Since service times of almost all procedures exported by RTS are short, we adopt the single threaded server model in RTS. However, there are several problems with implementing RTS using the model. Several functions supported by RTS may be blocked during execution. In this case, other requests may be suspended until the blocked procedure completes. Moreover, the server has a possibility of deadlock when using the model because other procedures may resume the blocked procedure. The advantage of the model is the simplicity of programming and the high performance. The model does not require many critical regions which may cause timing bugs which are difficult to remove.

In our solution, RTS creates several threads for processing requests, but all procedures are protected by one critical region as shown in Figure 7. The port attribute used in RTS specifies the FIFO policy for the dispatch policy, the BPI policy for the queueing policy and the HANDOFF ON policy for the priority handoff policy. The mutex attribute specifies the BPI policy for the queueing policy.

However, the measured results presented in the next section show that the cost of a process creation is very big. In this case, dispatching a thread in a server based on the priority may not be good idea. We may need to dispatch threads for processing an incoming request based on the procedure number as in ARTS[7]. The current version of RT-IPC does not include a procedure number in the messages. It makes it difficult to adopt the ARTS solution in Real-Time Mach.

### 4.2.2 Structure of NPS

Priority inversion in protocol processing may cause jitter when messages are periodically transmitted. Applications which expect a minimum jittering effect must the length of priority inversion to be very short. The static prioritized server model is most suitable for the server which supports such applications.

NPS has two sets of workers: one is for processing requests from users and another is for processing packets from a network as shown in Figure 8. Also, there is one manager thread which receives ethernet packets from a network. Since ethernet packets do not support the notion of priorities, the thread is

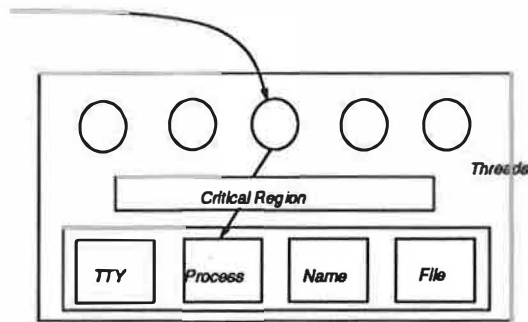


Figure 7: Structure of RTS

executed at the highest priority in the system<sup>2</sup>. Our version of IP supports priorities as a IP option. The ethernet driver and IP module are executed by the manager thread and the higher layers can be executed by input worker threads for the process of input packets. The manager thread passes a packet to an input worker thread according to the priority in an IP option. Output packets are processed by output worker threads.

In Real-Time Mach, the priority level is not small, for example, 32 levels in the fixed priority policy, then it is difficult to use the static prioritized model directly. The solution in NPS is to combine the static prioritized server model and the single threaded server model. If a request is received, a thread which will process the request is selected by the priority of the request. In contrast to the pure static prioritized server model, the priority of a request is inherited to a server. When the request is blocked and the priority of the request is higher than the priority of the thread which is dispatched by the priority of the request, the thread in the server inherits the priority of the request. The structure approximates the weakness of the pure static prioritized worker model. The port attribute for communicating a client and an output worker thread in NPS specifies the WORKER policy for the dispatch policy, the BPI policy for the queueing policy and the HANDOFF ON policy for the priority handoff policy. The port attribute for communicating the manager thread and an input worker thread needs to specify HANDOFF MSG policy as the priority handoff policy for inheriting the priority in a IP option to an input worker thread.

NPS enables users to configure the number of workers and their priority using a setup file, then application programmers can change the configuration by the characteristic of applications and enable to make the length of priority inversion short.

## 5 Evaluation

In this section, we show the performance of real-time servers. Before showing the performance, we present the costs of several basic functions in the Real-Time Mach kernel for analyzing the performance for RTS and NPS. We also show the effects of real-time servers by benchmark using RTS.

### 5.1 Basic Cost

We evaluated the several basic costs of the kernel primitives relating to real-time servers. We measured the costs in Gateway2000 486/33C which is an 33MHz Intel 80486 based IBM-PC compatible computer.

<sup>2</sup>We are working on FDDI driver because ethernet cannot provide bounded communication delay. Since the purpose of the paper is the structure of servers, we currently use ethernet driver in NPS.



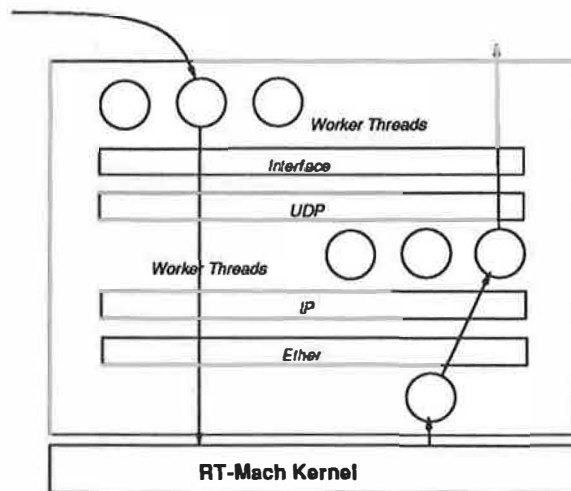


Figure 8: Structure of NPS

Original Mach ( $\mu$ s)	Mach Timesharing ( $\mu$ s)	Fixed Priority ( $\mu$ s)	Rate Monotonic ( $\mu$ s)
32	34	39	41

Table 1: Cost of Context Switch

The costs are measured on RTS under the fixed priority scheduling policy using the STAT! timer board whose granularity is 25 ns.

Table 1 shows the costs of context switching time between two threads in a same task under different scheduling policies. We also measured the context switch time of the original Mach kernel. The Mach-Timesharing policy is a policy which is used in the original Mach kernel. The implementation of the scheduler in Real-Time Mach is based on the concept of policy/mechanism separation. It enables users to change scheduling policies without rebooting or re-making a kernel. The measured times include the cost for blocking a current running thread, choosing a next runnable thread, enqueueing the blocked thread in a run queue, and context switching time. The difference in the costs under the Mach Timesharing policy on Mach and RT-Mach comes from the cost of the policy/mechanism separation. The difference in the costs under different scheduling policies comes from the implementation of run queues and priority compare functions. The fixed priority policy and the rate monotonic policy use priority queues and enqueueing requires comparing priorities of threads.

Table 2 shows the costs of creating new threads. Real-Time Mach can create an aperiodic thread and a periodic thread. The creation of a periodic thread includes the creation cost of a timer object for managing periodic execution. We also show the creation cost of a thread in the original Mach. There are two differences in the thread creations in Mach and Real-Time Mach. In RT-Mach, a program counter and a stack pointer are set in the creation call, but in Mach, the setup needs to call another system call. The creation call in Mach uses MIG, but the creation call in Real-Time Mach uses a trap call.

Table 3 shows the costs of allocating a new port. The allocation cost in Real-Time Mach depends on the number of buffers. We measured the cost when the number of buffers is 10 and the size is 2048 bytes, which are the values used in RTS. We also show the cost of allocating a port in the original Mach.

Table 4 shows the sum of the cost of *rt\_mutex\_lock* and *rt\_mutex\_unlock*. We measured the costs under different locking policies. The difference in the costs of the FIFO policy and the priority policy is

Mach thread ( $\mu s$ )	RT-Mach Aperiodic ( $\mu s$ )	RT-Mach Periodic ( $\mu s$ )
286	323	380

Table 2: Cost of Thread Creation

mach_port_allocate ( $\mu s$ )	rt_mach_port_allocate ( $\mu s$ )
67	162

Table 3: Cost of Port Allocation

caused by the overhead of using priority queueing in the priority policy. The difference in the costs of the priority policy and the basic priority inheritance(BPI) policy is caused by the overhead to store the priority information of the threads which acquire a lock.

Figure 9 shows the costs of the round-trip time in RT-IPC under different sizes of messages. We measured the costs by changing the queueing policy and, the priority handoff policy. In the measurement, we used FIFO policy for as the dispatch policy. We also show the cost of the round trip time in the original Mach IPC. The difference in the costs of the Mach IPC and the RT-IPC is caused by the cost to call the policy modules of the real-time resource manager. The difference in the costs under different policies is caused by the execution cost of each policy.

## 5.2 Evaluation of Real-Time Servers

Table 5 shows the performance in several primitives of two versions of RTS. The first version(RTS/RT-Mach) runs on RT-Mach, which uses the real-time thread package and the RT-IPC. The second version(RTS/Mach) runs on the original Mach, which uses the C-Thread package and the original Mach IPC. The table includes the costs to open a file, read 1024 bytes from a file, write 1024 bytes to a file, create a new process which does nothing, register a port and lookup a port. The results show that the difference in the costs of the two versions is bigger than the difference in the costs of RT-IPC and original Mach IPC and the additional cost by the mutex variable in Real-Time Mach. This fact means that other costs increase the performance more seriously.

We found two reasons which cause the differences. The first reason is the different implementation of *mig\_get\_reply\_port* in the real-time thread package(RT-Thread) and in the C-Thread packet(C-Thread). The function gets a port for a reply in the client side of the stub code generated by MIG. Many kernel calls in Mach use MIG and require reply ports even if the program does not call RPC to other programs or other machines. In multi-threaded applications, each thread needs a different reply port for calling the kernel. Since RTS uses MIG for communicating between clients and RTS, the function is called both in a client and in RTS. In C-Threads, the data structure which contains thread specific information is pointed to from the bottom of the stack. The data structure can keep a reply port for each thread. The solution can be used if the stack sizes of all threads are identical like C-Threads. However, in RT-Thread, each thread can have a different stack size. It is difficult to adopt the same solution used in C-Threads.

*Mig\_get\_reply\_port* in RT-Thread uses *mach\_thread\_self* to identify threads. Since the function is a system call, RPC calls using MIG (including kernel calls) in RT-Thread becomes more expensive than

FIFO ( $\mu s$ )	Priority ( $\mu s$ )	BPI ( $\mu s$ )
52	53	56

Table 4: Cost of Synchronization

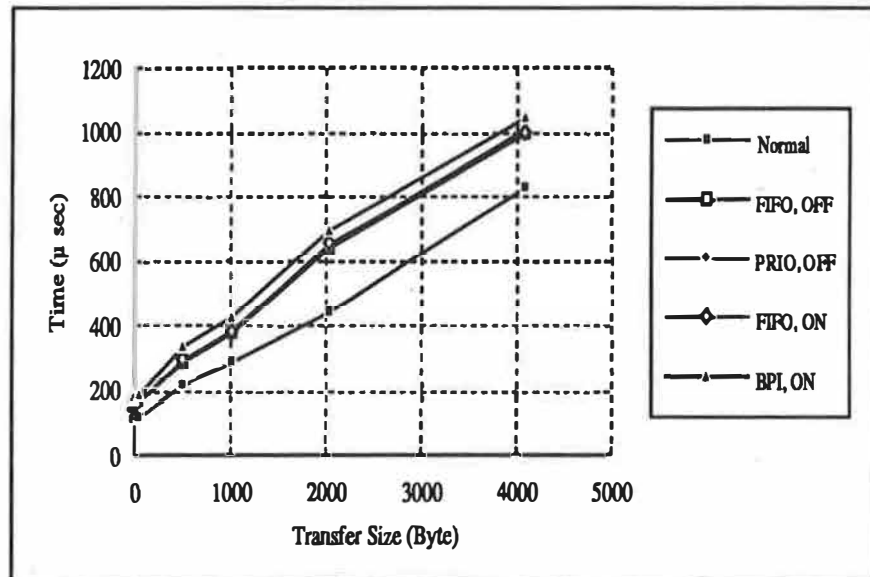


Figure 9: Cost of IPC

Function	RTS/RT-Mach (ms)	RTS/Mach (ms)
Open	1.39	0.95
Read 1024 bytes	3.09	2.85
Write 1024 bytes	1.76	1.50
Map 1024 bytes	1.12	0.79
Process Creation	34.67	33.52
Name Register	1.15	0.81
Name Lookup	0.90	0.63

Table 5: RTS Performance

in C-Thread. The implementation causes the major difference in the cost of RTS primitives between on RTS/RT-mach and RTS/Mach. In fact, the big difference in a process creation primitive of RTS/RT-Mach and RTS/Mach is caused by many kernel calls.

The second reason is the different cost in *rt\_mach\_port\_allocate* and *mach\_port\_allocate*. *Open* function for a file in RTS/RT-Mach allocates a new real-time port for identifying an opened file. The difference in the cost of a normal port and a real-time port is about 100  $\mu$ s. The big difference in the open functions in RTS/RT-Mach and RTS/Mach is caused by the difference in the costs of allocating ports.

The increased costs caused by RT-IPC and RT-synchronization are difficult to reduce, but the increased costs by them are about 10 % in each function. The problem is caused by the mismatch in the implementation of MIG and the current implementation of the real-time thread package.

Table 6 shows the performance of NPS which is running on RT-Mach(NPS/RT-Mach) and Mach(NPS/Mach). We measured the costs of the round-trip time between two tasks in different machines. In the measurement, the message sizes of request and reply are the same. We also show the costs of the round trip time using Unix socket interface on Mach2.5 and Mach3.0/UX.

Mach2.5 implements network protocols in kernel space. Packets from a network are processed in a

	Mach3.0/UX (ms)	Mach2.5 (ms)	NPS/RT-Mach (ms)	NPS/Mach (ms)
1 byte	7.50	2.01	3.30	2.50
100 bytes	7.80	2.61	3.95	3.41
1024 bytes	15.67	8.69	10.01	9.43
2048 bytes	23.42	14.32	15.72	15.04

Table 6: NPS Performance

soft interrupt handler. The approach does not have the cost of scheduling and context switch. NPS is implemented in a user space. User level network servers make the time to develop and debug protocols very fast. Since our focus is experiments with several protocols for real-time applications, the user level approach is more appropriate for our purpose. However, there are two reasons that increase the costs.

The first reason is that a user level network server requires the communication between a client and NPS. In NPS/Mach, the context switching overhead is very small, the major difference between in NPS/Mach and in Mach 2.5 socket is caused by the overhead by the communication between a client and NPS.

The second reason is the overhead which comes from the worker threads for processing incoming packets. The thread processing ethernet packets and the worker threads for processing incoming packets are different kernel threads and the threads communicate using RT-IPC to inherit the priority of a packet to a worker thread. The approach causes extra context switch time and communication overhead. The difference in the costs of NPS/RT-Mach and NPS/Mach is caused by this reason.

### 5.3 Effects of Real-Time Servers

We show two benchmarks which demonstrate the advantages of the real-time server model. In the benchmarks, the worst case is created artificially by setting a suitable start time for each thread. A real-time application should not miss its deadline even if the worst case occurs. The benchmarks demonstrate the difference in the traditional approach and in the real-time server approach when the worst case occurs.

The first benchmark shows the effect of the priority handoff from a client to a server. In the benchmark, a high priority client asks RTS to write 1024 bytes to a file 5 times every 100 ms, and we increase the execution time of a medium priority thread which is executed every 200 ms. We changed the queueing policy and the priority handoff policy of a port attribute and a mutex attributed in RTS. The first case uses the FIFO policy and the HANDOFF OFF policy. The second case uses the PRIO policy and the HANDOFF ON policy and the last case uses the BPI policy and the HANDOFF ON policy. We measured the break down CPU utilization<sup>3</sup> in three cases under the rate-monotonic policy. The results are shown in table 7. In the case of FIFO/HANDOFF OFF, the low priority thread misses its deadline, but the high priority thread misses its deadline in the cases of PRIO/HANDOFF ON and BPI/HANDOFF ON when the task set exceeds the break down utilization.

The second benchmark shows the effect of the basic inheritance protocol. In the benchmark, a low priority client is executed every 400 ms and a high priority client is executed every 100 ms. A high priority client asks to RTS to write 1024 bytes 5 times to a file and a low priority client asks to read 1024 bytes from a file 10 times. Also, we increase the execution time of a medium priority thread which is executed every 200 ms and measured the break down CPU utilization. We changed the queueing policy and the priority handoff policy and measured the break down CPU utilization like the first benchmark. In the case of PRIO/HANDOFF ON, requests are enqueued in a priority order. The priority of a thread in the server is bumped up by the priority inheritance protocol when a high priority thread is blocked in the case of BPI/HANDOFF ON. The results are shown in Table 7. The low priority thread misses its deadline in the case of BPI/HANDOFF ON and the high priority thread misses its deadline in the

<sup>3</sup>The deadlines of all threads are not missed if the CPU utilization is under the break down CPU utilization.

	Benchmark 1 (%)	Benchmark 2 (%)
FIFO/HANDOFF OFF	54	58
PRIQ/HANDOFF ON	99	62
BPI/HANDOFF ON	99	94

Table 7: Results of Benchmarks

cases of FIFO/HANDOFF OFF and PRIQ/HANDOFF ON if CPU utilization exceeds the break down utilization.

The results show the two advantages of real-time server model. The first advantage is increasing CPU utilization in the real-time server model. The second advantage is that a high priority thread missed its deadline in the real-time server model. This means that a high priority thread cannot miss its deadline even if the CPU utilization exceeds the break down CPU utilization and we can predict whether the deadlines of timing critical threads are ensured when we can know all worst case blocking times of the timing critical threads.

## 6 Related Work

ARTS[18, 7] is an object based distributed real-time operating system which has been developed at CMU. Many ideas in Real-Time Mach come from experiments with ARTS. The real-time server model is similar to the real-time object model in ARTS[7]. Real-Time Mach provides system support for the real-time server model. In ARTS, the worker model is adopted for processing network protocols[19], but users cannot use the model for their applications and there is no support for the integration of synchronization and IPC.

Sha[15] proposed several extensions of Ada for supporting the mechanisms for hard real-time systems. In Ada, tasks communicate with each other by rendezvous. The mechanism enables us to build single threaded server model. They propose to use the priority ceiling protocol in Ada rendezvous. The proposed model is very similar to the single threaded server model in Real-Time Mach. However, there is no support to create the worker model in Ada. It makes to write system servers such as network servers difficult.

Chorus[13] provides priority based preemptive scheduling and fast interrupt response. However, Chorus does not provide the mechanisms for providing predictability and analyzability due to the lack of the mechanisms for avoiding unbounded priority inversion. Many other commercial real-time operating systems, especially real-time Unix, also lack the mechanisms.

Synthesis[9] supports fine-grained adaptive scheduling for soft real-time systems. The scheduler in Synthesis is closely integrated with queues of I/O devices. If a queue of a device does not become empty, the scheduler tries to run a thread which processes messages in the queue. The policy makes the response time of I/O events very fast. The context switching time in Synthesis is very fast. Increasing the number of context switches does not cause big overhead.

In [10] and [22], they propose the extension of Mach micro kernel for multimedia applications. They support real-time scheduling, event notification mechanism, I/O management, and eager memory management.

## 7 Conclusion

In this paper, we presented the real-time server model which provides predictable services, the system supports for implementing the model and the evaluation of the implementation. We also described the experiments with RTS and NPS on Real-Time Mach.

The results in the paper show that predictability and analyzability have a cost. However, we found that the additional cost comes from not only the system support in RT-Mach but also the mismatch of

the implementation of existing Mach facilities such as MIG and the facilities in Real-Time Mach. We will continue further analysis for finding the sources of the overhead. They will help to develop better kernel support for real-time computing.

We are working on a user-level thread package which provides cheaper thread management than current kernel supported thread management. The user-level thread package enables us to change its semantics and syntax for the characteristic of applications and to remove the mismatch between existing tools and our system supports.

## Acknowledgments

We would like to thank the members of the ART Project and the Mach group for their valuable comments and inputs to the development of Real-Time Mach.

## References

- [1] D.P.Julin, J.J.Chew, J.M.Stevenson, P.Guedes, P.Neves and P.Roy, "Generalized Emulation Services for Mach 3.0: Overview, Experiences and Current Development." In Proceeding of USENIX 2nd Mach Symposium 1991.
- [2] D.Golub, R.Dean, A.Forin, and R.Rashid, "Unix as an application program", In the proceedings of Summer Usenix Conference, June, 1990.
- [3] D.Ferrari and D.Verma, "A scheme for Real-Time Channel Establishment in Wide-Area Networks", IEEE Journal of Selected Areas in Communications, Vol.8, No.3, 1990
- [4] J. P. Lehoczky and L. Sha, "Performance of Real-Time Bus Scheduling", ACM Performance Evaluation Review Vol.14, No.1, 1986.
- [5] J. P. Lehoczky, L. Sha, and Y. Ding. "The Rate Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behavior", In *Proceedings of the 10th IEEE Real-Time Systems Symposium*, December 1989.
- [6] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard real time environment", *Journal of the ACM*, Vol.20, No.1, 1973.
- [7] C.W.Mercer, and H.Tokuda, "The ARTS Real-Time Object Model", In Proceedings of 11th IEEE Real-Time Systems Symposium, 1990.
- [8] C.W.Mercer and H.Tokuda, "An Evaluation of Priority Consistency in Protocol Architectures", In Proceedings of the IEEE 16th Conference on Local Computer Networks, 1991.
- [9] H.Massalin and C.Pu, "Fine-Grained Adaptive Scheduling using Feedback", *Computing Systems*, Vol.3, No.1, 1990
- [10] J.Nakajima, M.Yazaki and H.Matsumoto, "Multimedia/Realtime Extensions for Mach3.0", Usenix Workshop on Micro-kernels and Other kernel Architecture, 1992.
- [11] T.Nakajima, T.Kitayama, H.Arakawa and H.Tokuda, "Priority Consistency Management in Distributed Operating Systems", In Preparation.
- [12] K. Loepere (ed), "Mach 3 Server Writer's Guide", Open Software Foundation and Carnegie Mellon University, 1992.
- [13] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemount, F. Herrmann, C. Kaiser, S. Langlois, P. Léonard, and W. Neuhauser, "Chorus distributed operating system", *Computing Systems Journal*, The Usenix Association, December, 1988

- [14] S.Savage and H.Tokuda, "RT-Mach Timers: Exporting Time to the User", In Proceeding of USENIX 3rd Mach Symposium, 1993.
- [15] L.Sha and J.Goodenough, "Real-Time Scheduling Theory and Ada", CMU SEI Technical Report, CMU/SEI-89-TR-14, 1989.
- [16] L. Sha, R.Rajkumar, and J.P.Lehoczky, "Priority Inheritance Protocols: An Approach to Real-Time Synchronization", IEEE Transactions on Computers, Vol.39, No.9, 1990.
- [17] A.S.Tanenbaum, "Modern Operating Systems", Prentice Hall, 1992
- [18] H.Tokuda and C.Mercer, "ARTS: A distributed real-time kernel", ACM Operating System Review, Vol.23,No.3,July,1989.
- [19] H. Tokuda, C. W. Mercer, Y. Ishikawa, and T. E. Marchok, "Priority inversions in real-time communication", In *Proceedings of 10th IEEE Real-Time Systems Symposium*, December, 1989.
- [20] H. Tokuda, T. Nakajima, and P. Rao, "Real-Time Mach: Towards a Predictable Real-Time System", In Proceeding of USENIX Mach Workshop, October, 1990.
- [21] H. Tokuda and T. Nakajima, "Evaluation of Real-Time Synchronization in Real-Time Mach", In Proceeding of USENIX 2nd Mach Symposium 1991.
- [22] J.A.Test, "Mach 3.0 Multimedia Real-Time Requirements", OSF Microkernel Workshop, 1992.





# Unix File Access and Caching in a Multicomputer Environment<sup>1</sup>

Paul J. Roy  
OSF Research Institute  
Cambridge, Massachusetts  
roy@osf.org

## Abstract

This paper describes the Unix file access and caching mechanisms in a version of the OSF/1 Unix operating system designed to run in a multicomputer environment. The multicomputer hardware platforms targeted can consist of hundreds or even thousands of individual nodes, where each node consists of one or more processors.

The multicomputer version of OSF/1 (called OSF/1 AD) uses Mach memory objects to cache data from Unix files, and relies on an in-kernel distributed shared memory implementation to maintain coherency for data cached across multiple nodes. The focus of this paper is on the modifications made to standard OSF/1 functionality to support distributed, efficient access to memory objects. Of particular interest are the introduction of a *mapped files module* for synchronizing clients and maintaining file meta data, the elimination of the traditional Unix buffer cache from the file data access path, and the implementation of a disk block reservation scheme to correctly support Unix write() semantics.

An evaluation of the technology is presented, providing insight into how it can be improved in the future, including several possible enhancements to Mach. As will be seen, most of this insight would equally apply to a single-node operating system based on Mach.

## 1 Introduction

Many computer vendors are now building computers utilizing a type of parallel processing known as Massively Parallel Processing (MPP). MPP computers, also known as *multicomputers*, can consist of hundreds or even thousands of nodes connected via a high-speed interconnect. Each node may contain one or more processors. In a typical multicomputer system (see Figure 1), the nodes of the multicomputer are divided into three groups: nodes used for input/output and connectivity (I/O nodes or file server nodes); nodes dedicated to parallel applications (compute nodes); and nodes for interactive use (service nodes). The number of nodes in each group may vary depending on the particular configuration.

The Open Software Foundation's Research Institute has built a multicomputer version of the OSF/1 operating system, called OSF/1 AD,<sup>2</sup> that provides a view of the hardware that looks like a conventional shared memory multiprocessor with an unusually large number of processors. The op-

---

<sup>1</sup>This research was supported in part by Defense Advanced Research Projects Agency (DARPA) and the Air Force Material Command (AFMC).

<sup>2</sup>OSF/1 AD is an acronym for OSF/1 with Advanced Development (AD) extensions from the OSF Research Institute.

erating system presents this notion of a Single System Image by building Unix functionality on top of base Mach services running on each node in the multicomputer.

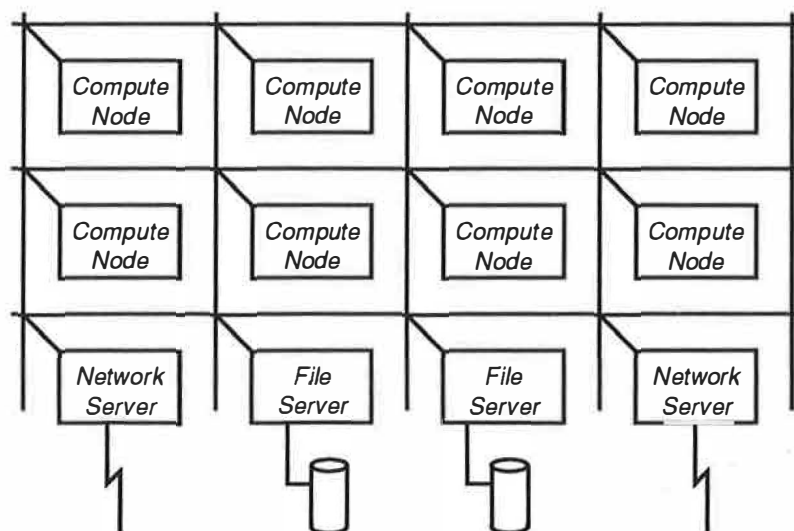


Figure 1: Multicomputer Architecture

This paper describes OSF/1 AD's Unix file access and caching mechanisms. It uses Mach memory objects to cache data from Unix files, and relies on an in-kernel distributed shared memory implementation to maintain coherency for data cached across multiple nodes. The focus of this paper is on the modifications made to standard OSF/1 functionality to support distributed, efficient access to memory objects. Of particular interest are the introduction of a *mapped files module* for synchronizing clients and maintaining file meta data, the elimination of the traditional Unix buffer cache from the file data access path, and the implementation of a disk block reservation scheme to correctly support Unix write() semantics.

An evaluation of the technology is presented, providing insight into how it can be improved in the future, including several possible enhancements to Mach. As will be seen, much of this insight would equally apply to a single-node operating system based on Mach.

After providing background information on OSF/1 AD and Mach memory objects, the remainder of the paper presents the goals and design of Unix file access, and evaluates the resulting technology. The paper concludes with sections on related work, implementation status, and ongoing work.

## 2 Background

### 2.1 OSF/1 MK Single Server

The Open Software Foundation's OSF/1 is an open operating system that incorporates advanced features while providing compatibility with industry standards and support for existing applications [18] [19]. OSF/1 is based on the Mach 2.5 operating system from Carnegie Mellon University, both of which are integrated or monolithic operating systems that implement the majority of traditional operating system functionality in the kernel.

The OSF/1 MK Single Server was created by dividing the OSF/1 monolithic kernel into the Mach 3.0 kernel and the OSF/1 MK Single Server. This was achieved by replacing OSF/1's Mach 2.5 internals with Mach 3.0, and adding a layer of compatibility code to allow the rest of OSF/1 to

execute as a user-mode server, the OSF/1 MK server. Among the features of the compatibility code are a threads library that provides lightweight user threads on top of Mach's kernel threads, and an *emulator* that implements some system functionality in application address spaces. This work was based on an earlier conversion effort at CMU involving the Mach 2.5 system [8]. The resulting OSF/1 MK server is pageable, preemptable, and multithreaded.

## 2.2 OSF/1 AD - A Multicomputer Version of OSF/1

OSF/1 AD extends the OSF/1 MK Single Server to a multicomputer environment [30]. Its architecture is derived directly from the target hardware architecture, in which large numbers of nodes are connected via a high-speed interconnect, and some subset of the nodes have peripheral devices attached. OSF/1 AD must allow all nodes and devices to be efficiently utilized and, in particular, must distribute OS functionality to avoid bottlenecks.<sup>3</sup>

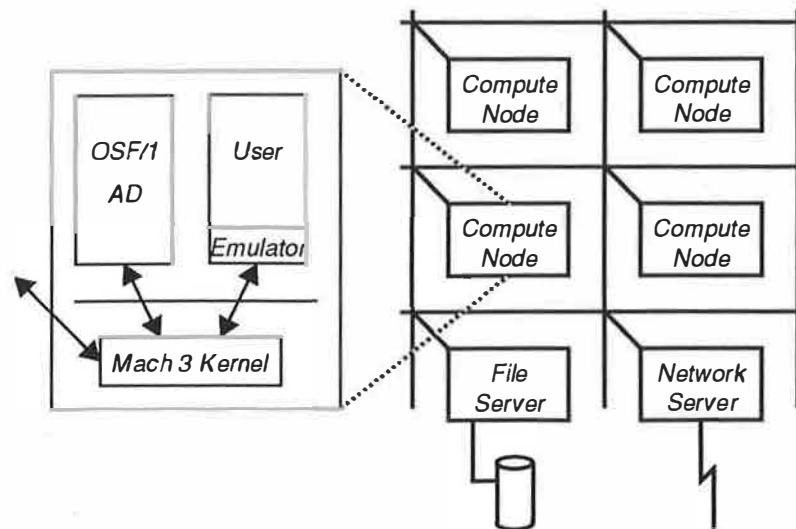


Figure 2: OSF/1 AD Architecture

These considerations lead to the following architectural model (see Figure 2):

- The Mach kernel runs on all nodes of the multicomputer, providing generic task/thread management, memory management, communication services, and device access.
- Specialized user-space servers implement Unix functionality, such as file service, process management, and networking.
- Disk and networking devices are managed by servers typically co-located on the same node with the device, although co-location is not mandatory.
- Process management functionality is distributed across most or all nodes on which application processes run.
- An *emulator* [8] [13] is available in each process's address space to provide some Unix functionality and perform system call-to-message conversion. It also contains a *callback thread* to receive messages from servers in support of interruptible system calls and file caching.

<sup>3</sup>Scalable process management functionality is provided by Transparent Network Computing (TNC) extensions from Locus Computing Corp.

- Process management system calls that require client-server interaction are converted by the emulator to messages to the process management server, which normally resides on the same node as the process. File management system calls that require client-server interaction are converted to messages to the file server managing the particular file being manipulated.

OSF/1 AD implements the entire OSF/1 Application Programming Interface (API). Although OSF/1 AD assumes no physical shared memory between nodes, it supports shared memory between processes residing on different nodes in software (see Section 2.4). Inter-node communication is provided by a modified version of the Mach network IPC presented in [1].

### 2.3 OSF/1 AD File System

OSF/1 AD extends the standard OSF/1 file system to support the integration of multiple file servers into a coherent whole. Each file server provides file service for one or more file systems (partitions), and mount operations are used to assemble multiple file systems across multiple file servers into a single file name space (Figure 3).

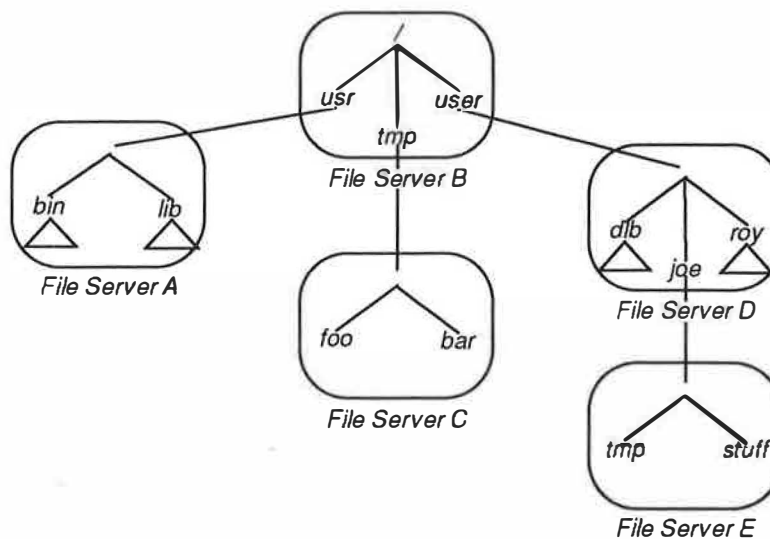


Figure 3: File Name Space

As in standard OSF/1, processes access files via system calls. In a multicomputer environment, it is extremely important to implement these system calls in a manner that minimizes IPC messages and server context switches. OSF/1 AD accomplishes this using the Mach system call redirection mechanism [8] to implement functionality in a per-process emulator residing in each process's address space. In particular, the emulator:

- contains the per-process file descriptor table, where each table entry has a Mach port right mapping to the open file structure on a file server. Hence, operations on open files communicate with at most one server.
- contains send rights to the Mach ports representing the root and current working directories for the process. Hence, most (but not all) operations on pathnames (e.g., `open()`) communicate with one server.
- provides a location to implement mapped access to open files and accrue the benefits of node-local file data caching. It is this aspect of the system that is the primary subject of this paper.

## 2.4 Mach Memory Objects

The address space of a Mach task is represented as a collection of mappings from addresses to offsets within memory objects. A Mach memory object represents a single source of memory (e.g., the file from which an executing program was loaded). The kernel manages physical memory as a cache of memory object contents; access to the actual memory (i.e., backing storage) is via a Mach port to which messages can be sent containing data or requesting that it be supplied [28] [29]. This allows memory objects to be implemented by user-state managers such as a file server or database application. The interface between the kernel and managers is known as the *External Memory Management Interface*.

To support access to memory objects on a multicomputer system, the XMM (eXtended Memory Management) subsystem has been added to the Mach kernel [2]. XMM supports access to memory objects from any node in the multicomputer, implementing a multiple-reader/single-writer policy for managing inter-node cache coherency. The policy is maintained on a per-page basis. The result is that pages from a single memory object may be cached across multiple nodes, with pages moved between nodes as necessary to maintain coherency in the face of ongoing memory object accesses throughout the multicomputer.

XMM is part of every Mach kernel on a multicomputer. Its implementation makes the collection of Mach kernels on a multicomputer behave as if they were a single kernel when communicating with memory object managers. This removes the complexity of distributed shared memory functionality from memory object managers. XMM also includes support for copy on reference between nodes for lazy evaluation of memory inherited between tasks on different nodes.

## 3 Goals

The primary goal of the Unix file access architecture was to enable efficient caching of file data on compute nodes. Achieving this goal would largely satisfy the performance and scalability requirements of a multicomputer file system because file data is frequently re-accessed by Unix applications.

Another goal was to support flexible allocation of memory cache resources to different types of data, depending on the behavior of applications. For example, a compute node running applications accessing large amounts of file data should have a much larger portion of its memory dedicated to cached file data versus, say, a compute node running applications with very large text and heap requirements.

Lastly, the design had to be amenable to a user-space implementation layered on top of the Mach kernel.

It is important to note that the time available for implementation was a constraint. For this reason, caching of other file system information, such as pathnames and attributes, was not implemented. Also, it was not a goal of this work to satisfy the needs of many supercomputer applications which require very high bandwidth access to vast amounts of data (but see Section 9).

## 4 Unix File Access and Caching

Because Mach memory objects have the ability to cache data provided by external, user-space managers, they are well suited to meet the goals outlined in Section 3. However, memory objects are only cache repositories; they don't provide full Unix file semantics. Hence, an additional level of functionality must be provided. Specifically:

**Client synchronization.** Each Unix read() and write() system call must be performed atomically with respect to other reads and writes of the same data.

**File meta data.** An open file's seek pointer, length, and its accessed and modified status must be maintained separately from the file's data.

**Cache control.** Cleaning and invalidating memory object-resident data must be done at appropriate times on behalf of some Unix system calls. For example, sync() must cause dirty file data to be written back to disk.

After describing the role of memory objects in Unix file access and caching, each of these items is discussed. In addition, changes to the underlying backing storage manager (the Unix File System) are presented. Figure 4 shows the relevant modules for this discussion.

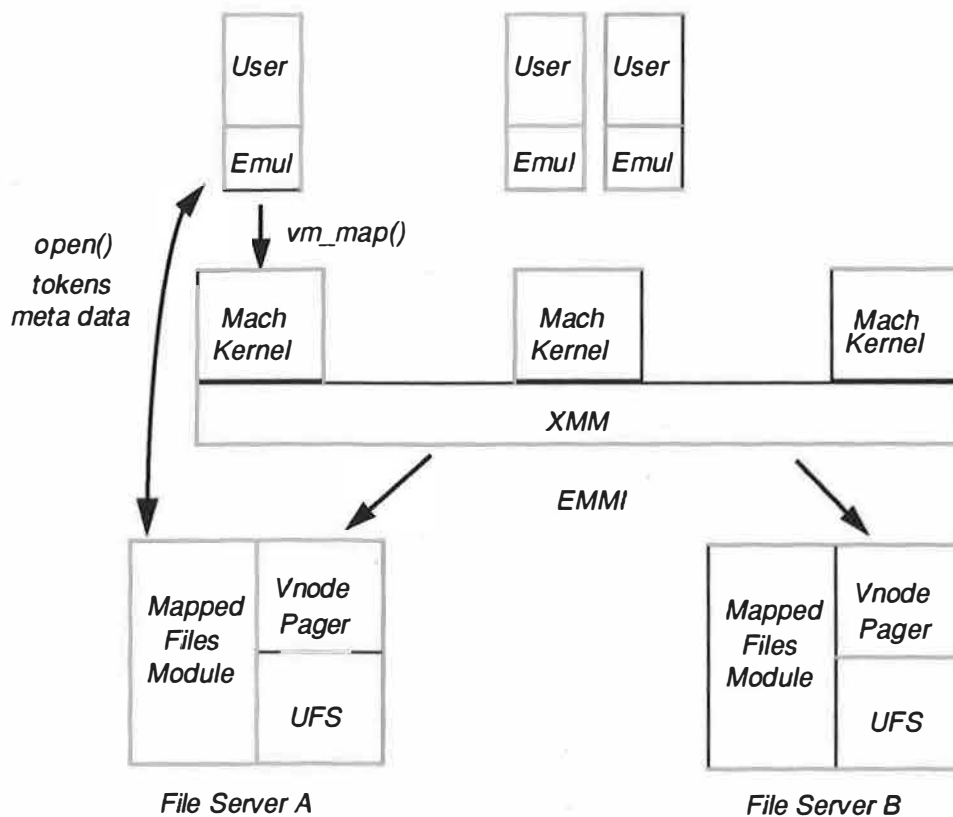


Figure 4: File Access and Caching Architecture

#### 4.1 The Role of Memory Objects

A Mach memory object is used to cache data from a regular Unix file in the following manner:

- A file's memory object port is obtained at open() time from the file server and stored in the emulator's file descriptor table.
- read() and write() system calls are converted by the emulator to mapped accesses to the memory object. These accesses are accomplished by mapping the memory object port, and using bcopy() to access the mapped region.
- Cache hits to already-resident data are satisfied without file server interaction. Also, because mapped regions are retained across successive Unix system calls, these cache hits may not involve any page faults or TLB misses.

- Cache misses are handled by the Mach kernel using the External Memory Management Interface to request data from the file server.
- The memory object cache is effectively “write-back.” The Unix sync() system call causes dirty data to be written back to the file server (again via the External Memory Management Interface), which in turn writes it to disk.
- Mach’s XMM kernel subsystem provides inter-node cache coherency of file data (XMM is discussed in Section 2.4).

Note that the Mach 3.0 4.3BSD Single Server and Chorus SVR4 systems [3] [6] have implemented similar file caching schemes for single-node systems.

## 4.2 Client Synchronization

Implementing POSIX file I/O semantics [10] requires that read() and write() system calls be performed atomically with respect to other reads and writes of the same data. In a single-node environment, this has historically been provided by a low-level file system lock (e.g., the inode lock). In a distributed environment, another mechanism is needed for synchronizing clients, while also minimizing the amount of necessary communication.

OSF/1 AD implements a module, known as the *mapped files module* (not to be confused with mmap() functionality), for synchronizing distributed clients. This module exports the notion of a *token*, which provides its holder the right to access a particular file’s data with either read or read/write capability. An instance of the mapped files module is implemented within each file server and supports the files managed by that server. Message-based interfaces are provided for acquiring and releasing tokens.

A client emulator wishing to access a file must first acquire a token from the file’s file server (with read or read/write capability), at which point access to the corresponding memory object may proceed. Subsequent I/O’s to the same file may find the token still held. However, it is also possible the token has been revoked, in which case a message to the file server is necessary to reacquire a token.

A file server revokes a token by sending a message to an emulator’s callback thread, instructing it to release the token. Token revocation occurs whenever an emulator attempts to acquire a token specifying a capability conflicting with already outstanding tokens. Other reasons for revoking tokens are associated with file meta data (see Section 4.3).

Tokens are implemented using Mach ports. In the event of an abnormal process abort, in which case tokens may not be released properly, file servers will receive no-more-senders notifications from the kernel [14] for all outstanding tokens, allowing them to perform necessary clean up.

## 4.3 Maintaining File Meta Data

In addition to synchronizing client data access, tokens are used to implement caching of file meta data to enable node-local file data caching, while still correctly implementing read() and write() semantics. Specifically, possession of a token allows a client emulator to cache an open file’s:

- seek pointer
- length
- accessed and modified status

The seek pointer determines the offset within the memory object to read or write. The length is necessary to implement append-mode writes, and to prohibit reading beyond the end-of-file. Re-

taining whether a file has been read or written is necessary so that proper accessed and modified times may be recorded.

When a token is released by an emulator (either voluntarily or because of a revocation), the cached state associated with the token is written back to the file server. If an application closes a file, the application's emulator voluntarily releases the token.

There are cases when a file server itself needs access to state that may be cached. For example, `truncate()` requires exclusive access to the file and its length, `sync()` must determine if there is modified file data to be written back to disk, and `stat()` must determine whether the file has been accessed or modified so that proper times may be reported. The mapped files module supports such cases by exporting well-abstracted interfaces to the rest of the server's file system code, thus hiding the fact that the implementation is revoking tokens as necessary.

Note that POSIX semantics [10] for updating a file's accessed and modified times only require that they be updated during the last `close()` of the file, or if the file's attributes are read (due to `stat()` or `fstat()`). Thus, emulators retain booleans indicating whether a file has been accessed or modified, relying on file servers to revoke tokens and update times as necessary. In addition to the requirements specified by POSIX, `sync()` also revokes tokens so that times may be updated. This results in behavior equivalent to a traditional UFS where booleans are similarly maintained in inodes.

Ideally, caching a file's meta data would support a multiple-reader/single-writer policy for accessing the file's memory object, with the exception of the seek pointer which necessitates a mutual exclusion policy for all processes sharing an open file. However, at the current time, the mapped files module's implementation is not yet optimized to distinguish these cases, resulting in mutual exclusion for all processes accessing a file's memory object. Exclusion lasts for the duration of time it takes to access/update the memory object on behalf of any given `read()` or `write()`. (This problem is being rectified as part of ongoing work - see Section 9).

#### 4.4 Memory Object Cache Control

Although Mach memory objects are used to hold cached file data, *most* of the control of the cache is the responsibility of file servers. In particular, a file server will invalidate cached data on behalf of a `truncate()` system call, and cause dirty pages to be written back to disk on behalf of a `sync()` or `fsync()` system call. In addition, a file server may choose to mark a memory object as "temporary," preventing write-backs of dirty data to the file server when the memory object is no longer active. When unlinking files, this technique avoids write-backs of data that will just be deallocated anyway.

This server-based cache control is performed by the mapped files module in close cooperation with another file server module, the *vnode pager*. The vnode pager interfaces directly with the Mach External Memory Management Interface.

Cache control as it relates to inter-node cache coherency of file data is the responsibility of the XMM kernel subsystem. It implements a multiple-reader/single-writer policy on a per-page basis. Also, per-node page cache management (e.g., page reclamation) is still under the jurisdiction of the local Mach kernel.

#### 4.5 Enhancements to the Unix File System (UFS)

The Unix File System (UFS) is used to provide disk backing storage for the system. Some important changes have been made to the standard OSF/1 UFS to provide better performance and correct behavior in OSF/1 AD.



**No-buffer-cache.** A performance-motivated change was the elimination of the traditional Unix buffer cache from the access path for file data. The buffer cache is still used for directory and meta data. Thus, when file data is read or written through the UFS, the UFS provides allocation of disk blocks and translation from logical file offsets to physical disk block addresses. The UFS provides no file data caching.

This *no-buffer-cache* enhancement was made for several reasons. First, because memory objects already cache data, it avoids a double caching effect. Second, it eliminates data copies into and out of the buffer cache. Third, it supports a mode of file access where data caching of any sort, even memory object caching, is undesirable. This mode is useful for satisfying the high bandwidth requirements of many supercomputer applications (see Section 9).

Besides data caching, the traditional buffer cache provides other functionality, namely read-ahead and write-behind. In OSF/1 AD, these functions were implemented and appropriate synchronization provided to avoid access to stale data. This involves blocking read requests while conflicting asynchronous disk writes are in progress because the kernel doesn't provide ordering guarantees, and synchronizing incoming write requests with the read-ahead mechanism. An important distinction between this read-ahead mechanism and that provided by the traditional buffer cache is that a hit on read-ahead data actually removes the data from the read-ahead cache, thus avoiding the unwanted copy and double caching effect.

Another important aspect of this design is that it will enable the UFS to coalesce contiguous file system blocks into larger disk reads and writes, thus obtaining greater disk bandwidth. Other systems have accrued this same benefit via extents [5], or extent-like mechanisms [16]. This optimization has not yet been implemented.

**Disk block reservation.** When `write()` is called, the system must guarantee that enough disk space exists to satisfy the write, and if not, return an `ENOSPC` error. For newly written data (i.e., no disk space yet exists), a memory object cache miss is guaranteed to occur, which provides the file server with an opportunity to allocate disk space. However, the UFS can't simply allocate disk blocks and enter them into the inode map because a subsequent crash would make those blocks appear to be valid portions of the file when in fact they don't yet contain valid data.

This problem necessitates some form of disk block reservation. The possibility of simply decrementing the free block count for the partition has appeal, but was rejected because of the inability to determine how much to add back to the free block count when a file with holes is truncated. Also rejected was the use of in-core data structures to keep track of all reserved blocks for all files, because of their potential to grow very large, and because of the desire to avoid allocating additional memory resources at a time when the system is already under memory pressure due to data being written.

The solution chosen relies on stealing the high bit of the disk address (`daddr`) fields stored in inodes to indicate that the rest of the `daddr` field represents a "reserved" disk address. In other words, setting this bit indicates that the disk address is valid but the corresponding disk blocks do not yet contain valid data. Hence, at the time of `write()`, disk addresses are reserved, but then converted to real disk addresses (by masking off the high bit) when the data is actually written back from the kernel to the UFS.

This scheme required a couple hundred lines of additional code to the UFS, but they are very well contained within the allocation and truncation routines. A few lines of changes were also needed to `ufs_fsck` so that it would treat reserved disk addresses as invalid when the partition is salvaged after a crash.

Since dadddr's address file system fragments, this change decreases the maximum amount of addressable fragments in a partition. But, conservatively assuming a 1 Kb fragment size, this results in the maximum partition size decreasing from 4 terabytes to 2 terabytes, which is tolerable, especially given that this limit can be overcome by either using a larger fragment size or partitioning a disk into more partitions.

## 5 Evaluation

This section evaluates the system, providing insight into how it can be improved in the future, including several possible enhancements to Mach. The particular areas addressed are:

- Mach's External Memory Management facility.
- The mechanisms provided by Mach for accessing memory object data.
- Variable-sized caching for different types of data.
- Inter-node coherency of file data.
- The mapped files module.
- Read-ahead for memory object-based files.
- The emulator.

### 5.1 External Memory Management

The External Memory Management functionality provided by Mach has proved to be well suited to a modular file system implementation. Of particular note is the fact that caching functionality can be cleanly separated from management of backing storage. Alternative designs where caching and backing storage are tightly integrated can result in intricate virtual memory/file system interaction [16], and are not well suited to a user-space implementation of backing storage managers [3].

An effect of the clean separation is that it necessitates an extension to the backing storage manager (in this case, UFS) to reserve disk blocks so that correct ENOSPC semantics for the write() system call can be implemented. However, these changes are quite well encapsulated and are expected to also be useful for implementing pre-allocated and real-time files.

A difficulty with using the External Memory Management functionality is maintaining the correct length of a Unix file. Because Mach memory objects do not have a notion of length, write-backs of data from the kernel to the backing storage manager are of page size granularity. As described previously, the OSF/1 AD file system is able to maintain the exact length (in bytes) separately, but not without complexity.

In particular, when a file is being grown due to the write() system call, the memory object and length of the file must be updated atomically by the emulator. Unfortunately, while a memory object is being updated it is possible for the page(s) being modified to be written back to the manager (e.g., due to high memory demand from other applications), blocking the thread performing the update. The manager, in turn, may then have to send a callback to the emulator to determine the exact length of the file so that it may write the proper amount to disk. To avoid deadlock, the execution of this callback must avoid blocking on the blocked thread, but doing so violates the atomicity requirement of the file grow operation.

The solution chosen for this problem allows the atomicity violation to occur, and because the emulator updates the file length before the memory object, can result in the length of the on-disk file temporarily being too large relative to the amount of valid data it contains. However, the system

guarantees that the page(s) in question will be written back again, this time containing enough valid data so that the length of the on-disk file accurately reflects the amount of valid data.

In practice, this solution works because 1) the chances of a page being written back while it is being modified is unlikely, and 2) even then, a crash would have to occur to expose the invalid data (zero's) at the end of the file to applications. Nonetheless, additional support at the Mach level to prevent pages from being written back while they are being modified would eliminate the problem altogether and avoid much complexity in the Unix layer. Although Mach's `vm_wire()` call [14] could be used to wire the pages involved, a cheaper mechanism is needed (such as the one proposed in Section 5.2)

Perhaps the most attractive solution, from the perspective of Unix file access, would be for Mach to maintain a notion of *length* of memory objects, thus enabling write-backs of data on a byte-size granularity and reducing the burden on the Unix layer to maintain the exact length of cached data. However, this represents a fundamental change to the Mach memory object abstraction that must not be taken lightly.

## 5.2 Memory Object Access

The only way to access a Mach memory object is to map it and access the mapped region (e.g., via `bcopy()`). Although there is some overhead associated with mapping, OSF/1 AD maps sufficiently large windows into memory objects to amortize the cost across many (smaller sized) `read()` and `write()` system calls. And, once a particular page has been faulted in and installed in the TLB, subsequent accesses to the same page can be very fast because faults and TLB misses are avoided.

Unfortunately, accessing memory objects via page faults has a serious problem: information is lost across the Mach boundary. With respect to Unix file access, this problem is manifested in the following ways:

**Size of reads/writes.** Because Mach treats all page faults as a request for one page, it has no knowledge of the amount of data actually being read/written. In the case of reading, it is thus unable to request larger amounts from the backing storage manager. Although the fault path *could* guess and read-ahead, it is suboptimal (consider large, non-sequential reads). Mach's inability to intelligently "cluster" data can have a significant performance impact due to extra disk I/O's.

Note that standard OSF/1 has implemented a form of clustering [4], where Mach's notion of cluster size is set on a per-object basis by the file system, using the External Memory Manager Interface. This is sufficient for access to anonymous memory where the optimal cluster size is mainly a function of the optimal disk access time, and hence is known to the file system at the time the object is initialized. It is also sufficient for access to `mmap()` regions where the cluster size can be initialized and later changed based on advice from the Unix `madvise()` system call.

However, for Unix file access the cluster size should be a function of the size of the particular Unix `read()` or `write()` system call in progress. But, changing the cluster size via the External Memory Manager Interface during the execution of `read()` or `write()` would be prohibitively expensive, especially if the application is remote from the file server. And, it would suffer from the inability to handle the case of multiple applications simultaneously accessing a memory object, each with a different optimal cluster size.

Also, in the common case of growing Unix files, the associated write faults will make one request to the backing storage manager for each page (to find out if data for the page exists, and if not, zero-fill it). Many of these requests could effectively be "batched" if Mach was able to contact the backing storage manager on behalf of larger sizes.

**Error returns.** There is no way to return error codes from a page fault. This is especially problematic for implementing ENOSPC semantics of the write() system call, which requires that an ENOSPC error code be returned if there is no more disk space left to satisfy the write(). OSF/1 AD handles this by passing ENOSPC to the memory\_object\_data\_error() kernel interface, which in turn generates an exception. The exception handler in the file server then modifies the program counter of the thread blocked accessing the memory object, forcing it to “jump” to a new location to pick up an ENOSPC error code. The location to jump to must have previously been “registered” with the exception handler. Obviously, a simpler mechanism would be preferable.

**Type of access.** The fault path is unable to distinguish between faults due to read() and write() versus those due to accesses to mmap() regions. A key difference in semantics for these cases is that writes beyond the end-of-file should grow the file whereas attempts to update mmap() regions beyond the end-of-file should not. Barring changing mmap() semantics (to prevent mapping a file beyond the end-of-file), an efficient solution to this problem is difficult (if not impossible) without additional information regarding the type of access.

These examples justify experimentation with new Mach interfaces for explicitly copying data to and from memory objects. These interfaces would provide the necessary channels through which access size and other information could be passed. They could also be brought to bear on the problem with file length handling described in Section 5.1 because the pages involved could be temporarily wired while they are being accessed. An important aspect of this experimentation would be to quantify the negative performance impact of the resulting additional Mach system calls.

### 5.3 Variable-Sized Caching

Because memory objects are used for caching both file data and program data (text, heap, etc.), the amount of memory allocated to the different types of data can vary, depending on the particular load. The Sprite system has demonstrated the benefit of such variable-sized caching [17]. Indeed other systems are unifying caching in this way [3] [7].

A problem is that Mach treats all memory equally when making page reclamation decisions, with the exception of deactivating-behind pages (making them better candidates for reclamation) from sequentially accessed memory objects. And, in all cases, Mach doesn't know whether pages belong to a memory object containing file data versus a memory object containing other types of data. This precludes biasing reclamation decisions based on the type of data.

There are several approaches to this problem. One is to provide knowledge to the kernel about the type of data and/or access behavior, but still allow it to make the ultimate page reclamation decisions (the Sprite system falls in this category). Other approaches allow external managers to affect all page management decisions to varying degrees [9] [15] [23]. For the particular problem at hand, providing additional information to the kernel (e.g., type of object, expected access behavior) so that it may make more intelligent page reclamation decisions is probably sufficient.

### 5.4 Inter-Node Coherency of File Data

XMM has proved adequate as a mechanism for providing inter-node coherency of file data. An aspect of this approach, however, is that XMM only has semantic knowledge of memory objects. While this is often sufficient for distributed shared memory implementations, its lack of information regarding Unix file sharing behavior can have a performance impact. In particular, when a file is cached in one node and, say, written on another node, XMM is able to move pages between nodes but does so one page at a time while they are faulted on.

A possible optimization is to provide inter-node cache coherency at the Unix layer. It would be a straightforward extension to perform appropriate flushing or invalidation as part of the token re-

voke mechanism, thus accruing the benefits of operating on multiple pages at a time and avoidance of XMM machinery. Currently, though, flushing or invalidating data can only be done by the file server because it holds the control port for the memory object, which would involve extra messages. A new Mach interface would be needed to support this type of cache control using only the memory object port.

Also, XMM is “involved” for all memory object messages whether it is needed or not. Although this provides multi-node transparency from the perspective of backing storage managers, there is some cost for objects not needing its capabilities. Indeed, given that most Unix files are *not* shared, a design that assumes all files *are* shared seems suboptimal. Further investigation is needed to quantify the costs imposed by XMM.

A last note is that XMM is not yet able to process multi-page write-backs of data. As a result, many write-backs to backing storage are artificially broken up into single page units, thus impacting disk write bandwidth. This must be rectified in future versions of XMM (see Section 9).

## 5.5 Mapped Files Module

The mapped files module hides the implementation of tokens, cached file meta data, cache control, etc. from the rest of the Unix code via well-abstracted interfaces. For example, there are interfaces for cleaning, truncating, getting a file’s size, making a file temporary, etc. For implementation expediency, invocation of its interfaces are sprinkled in code that is otherwise unmodified from standard OSF/1 code. But, because the semantics of the mapped files module is effectively a caching module layered *above* the disk storage manager, it appears well-suited to a stacked vnode implementation [25]. In fact, the current implementation is already quite well layered and should easily transition to a stacked implementation.

At the current time, the mapped files module only supports mutually exclusive access to a file’s memory object. This must be optimized to support a multiple-reader/single-writer policy.

## 5.6 Read-ahead

As described previously, the UFS no-buffer-cache code implements read-ahead so that the disk wait time for subsequent data requests from the kernel can be reduced or eliminated. Although Unix has historically performed read-ahead at this layer, it is suboptimal for a variety of reasons.

First, it is making policy decisions about *when*, *where*, and *how much* to read-ahead, although such decisions could more flexibly be provided at a higher layer in support of application-specific read-ahead policies. Second, initiating reads from deep within the UFS is problematic for systems needing to intercept all reads, such as hierarchical storage systems (see [26]). Third, the UFS doesn’t have knowledge of what data is already cached, and hence may perform wasted read-aheads. Fourth, because memory object cache hits do not involve any interaction with the UFS, no read-aheads will be initiated in these cases. Fifth, even when data is found in a read-ahead buffer the latency for the application can be quite long, especially if the application is remote from the file server.

All of these deficiencies can be rectified by initiating read-aheads from a higher layer and reading the data directly into memory objects. The kernel would perform such reads (using the External Memory Manager Interface) but the policy of *when*, *where*, and *how much* to read-ahead could be left to a layer above the kernel. Again, such a scheme would require that additional information be provided to Mach, perhaps as extension to the explicit memory object access interfaces suggested in Section 5.2.

## 5.7 Emulator

The main benefit of the emulator is that it is a place to implement system functionality “near” the application (i.e., it is quick to get to when an application makes a system call). However, it has problems. One is that it has become quite large which, because it is per-process, has memory consumption and process creation overhead ramifications. But, perhaps most importantly, its unprotected nature results in a fair amount of complexity and overhead in servers to “defend” themselves against the possibility of corruption in the emulator’s state, and often results in an artificial partitioning of code and state based on protection concerns rather than real modularity concerns.

Future revisions of the OSF/1 AD system will utilize an alternative to the emulator where system calls are redirected to a local server using the exception mechanism [22].

## 6 Relevance to Single-Node Systems

Although OSF/1 AD is targeted for multicomputers, most of the work described in this paper would equally apply to a single-node operating system based on Mach. In fact, the entire evaluation (Section 5) is relevant, excluding the discussion on inter-node data coherency (Section 5.4). And, the interfaces to the mapped files module would remain largely unchanged, although its implementation could be simplified, especially in a system without the emulator (see Section 5.7) because all file access would be centralized in the server, thus obviating the need for tokens.

A key aspect of OSF/1 AD’s file caching design is the clean separation between caching functionality and management of backing storage. This separation would equally apply to a single-node operating system, and would result in a more modular system that should enable easier integration of new backing storage managers.

## 7 Related Work

The Mach 3.0 4.3BSD Single Server and Chorus SVR4 systems [3] [6] have implemented Unix file caching schemes based on unifying the cache management of file data and program data, although these systems are restricted to single-node environments. SunOS [7] also unifies caching but is restricted to a single-node, monolithic kernel environment.

Some early work on using Mach memory objects for Unix file caching was done in previous Mach systems [24].

Sprite [21] is a research system designed for a network of workstations. It provides a single file system image throughout all nodes in the cluster [27], and unifies the cache management of file data and program data [17].

Multics [20] and Apollo DOMAIN [13] are early examples of non-Unix operating systems providing the notion of a single level store, where access to all data is by mapping objects. Multics runs on single-node systems whereas DOMAIN runs across a network of workstations.

## 8 Implementation Status

OSF/1 AD is the base for the operating system for the Intel Paragon XP/S Supercomputer [11]. Typical configurations of the Paragon range from fewer than 100 nodes to more than 1000 nodes. Additionally, OSF/1 AD runs on clusters of i386 and i486 PC’s on an Ethernet, simulating a multicomputer.

All of the work discussed in this paper is available from the OSF Research Institute in its OSF/1 AD prototype which is available as part of the standard OSF/1 license.

## 9 Ongoing Work

All of the suggested improvements in the evaluation (Section 5) are being pursued, with initial emphasis on enhancing the mapped files module to support a greater degree of concurrency, modifying XMM to enable processing of multi-page write-backs, and improving the memory object access and variable-sized caching mechanisms. In addition, future revisions of the architecture will not have an emulator and the mapped files module will be better integrated with the rest of the standard OSF/1 code.

Performance measurement and tuning is also underway. An accurate performance evaluation of the overall architecture will not be possible, however, until the system is able to perform multi-page data transfers across the External Memory Management Interface (see Sections 5.2 and 5.4).

There is also work in progress to support the needs of many supercomputer applications which require very high bandwidth access to vast amounts of data. A file access path bypassing all data caching will be provided, leveraging the existing UFS no-buffer-cache code.

## 10 Conclusion

This paper has presented the Unix file access and caching architecture in OSF/1 AD, a version of the OSF/1 operating system designed to run in a multicomputer environment. OSF/1 AD uses Mach memory objects to cache data from Unix files, and relies on an in-kernel distributed shared memory implementation to maintain coherency for data cached across multiple nodes.

The focus of the paper was on the modifications made to standard OSF/1 functionality to support distributed, efficient access to memory objects. These modifications include the introduction of a mapped files module for synchronizing clients and maintaining file meta data, the elimination of the traditional Unix buffer cache from the file data access path, and the implementation of a disk block reservation scheme to correctly support Unix write() semantics.

A key aspect of OSF/1 AD's file caching design is the clean separation between caching functionality and management of backing storage. This separation results in a more modular system that should enable easier integration of new backing storage managers.

An evaluation of the system has identified many possible areas for improvement. Suggested enhancements being pursued, include:

- New Mach interfaces for explicitly copying data to and from memory objects.
- More intelligent page reclamation decisions within Mach.
- Provision of inter-node cache coherency of file data in the Unix layer rather than relying on Mach's XMM functionality.
- An improved read-ahead scheme for memory object-based Unix files.

Although OSF/1 AD is targeted for multicomputers, most of the work described in the paper, including the suggested enhancements, would equally apply to a single-node operating system based on Mach.

## 11 Acknowledgments

Paul Dale, George Feinberg, and Alan Langerman provided useful comments on the contents and presentation of this paper.

## References

- [1] Barrera, J. A Fast Mach Network IPC Implementation. In *Proceedings of the USENIX Mach Symposium*, November 1991.
- [2] Barrera, J. Copying, Caching, and Sharing in a Distributed Virtual Memory System. Ph.D. dissertation, Carnegie-Mellon University, to appear 1993.
- [3] Batlivala, N., Gleeson, B., Hamrick, J., Lumdal, S., Price, D., Soddy, J., Abrossimov, V. Experience with SVR4 over Chorus. In *Proceedings of the USENIX Workshop on Micro-Kernels and Other Architectures*, April 1992.
- [4] Black, D., Carter, J., Feinberg, G., MacDonald, R., Mangalat, S., Shienbrood, E., Van Sciver, J., Wang, P. OSF/1 Virtual Memory Improvements. In *Proceedings of the USENIX Mach Symposium*, November 1991.
- [5] Cheriton, D., Roy, P. Performance of the V Storage Server: A Preliminary Report. In *Proceedings of the 1985 ACM Computer Science Conference*.
- [6] Dean, R., Armand, F. Data Movement in Kernelized Systems. In *Proceedings of the USENIX Workshop on Microkernels and Other Architectures*, April 1992.
- [7] Gingell, R., Moran, J., Shannon, W. Virtual Memory Architecture in SunOS. In *Proceedings of the Summer 1987 USENIX Conference*.
- [8] Golub, D., Dean, R., Forin, A., Rashid, R. Unix as an Application Program. In *Proceedings of the Summer 1990 USENIX Conference*.
- [9] Harty, K., Cheriton, D. Application-Controlled Physical Memory using External Page-Cache Management. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. October, 1992.
- [10] IEEE. Portable Operating Systems Interface (POSIX) - Part 1: System Application Program Interface. IEEE Std 1003.1, 1990. ISO/IEC 9945-1.
- [11] Intel Corporation. Intel Paragon XP/S Supercomputer Spec Sheet. 1992.
- [12] Julin, D., Chew, J., Stevenson, M., Guedes, P., Neves, P., Roy, P. Generalized Emulation Services for Mach 3.0 - Overview, Experiences, and Current Status. In *Proceedings of the USENIX Mach Symposium*, November 1991.
- [13] Leach, P., Levine, P., Hamilton, J., Stumpf, B. The File System of an Integrated Local Network. In *Proceedings of the 1985 ACM Computer Science Conference*.
- [14] Loeper, K. (editor). *Mach 3 Kernel Interfaces*. Open Software Foundation and Carnegie Mellon University.
- [15] McNamee, D., Armstrong, K. Extending the Mach External Pager Interface to Accommodate User-Level Page Replacement Policies. In *Proceedings of the USENIX Mach Symposium*, October, 1990.
- [16] McVoy, L., Kleiman, S. Extent-like Performance from a Unix File System. In *Proceedings of the Winter 1991 USENIX Conference*.
- [17] Nelson, M. Physical Memory Management in a Network Operating System. Ph.D. dissertation, University of California at Berkeley, November 1988. UCB/CSD 88/471.
- [18] O'Reilly & Associates, Inc. Guide to OSF/1: A Technical Synopsis. 1991.



- [19] Open Software Foundation. The Design of the OSF/1 Operating System. 1990.
- [20] Organick, E. The Multics System: An Explanation of Its Structure. M.I.T. Press, 1972.
- [21] Ousterhout, J., Cherenon, A., Douglass, F., Nelson, M., Welch, B. The Sprite Network Operating System. In *Computer*, February 1988.
- [22] Patience, S. Redirecting System Calls in Mach 3.0: An Alternative to the Emulator. In *Proceedings of the USENIX Mach Symposium*, April 1993.
- [23] Sechrest, S., Park, Y. User-Level Physical Memory Management for Mach. In *Proceedings of the USENIX Mach Symposium*, November 1991.
- [24] Tevanian, A. Architecture-Independent Virtual Memory Management for Parallel and Distributed Environments: The Mach Approach. Ph.D. dissertation, Carnegie Mellon University, December 1987. CMU-CS-88-106.
- [25] Unix International, Stackable Files Work Group. Requirements for Stackable Files. Revision 3.6, February, 1993.
- [26] Webber, N., Operating System Support for Portable Filesystem Extensions. In *Proceedings of the Winter 1993 USENIX Conference*.
- [27] Welch, B. Naming, State Management, and User-Level Extensions in the Sprite Distributed File System. Ph.D. Dissertation, University of California at Berkeley, April 1990. UCB/CSD 90/567.
- [28] Young, M., Tevanian, A. Rashid, R., Golub, D., Eppinger, J., Chew, J., Bolosky, W., Black, D., Baron, R. The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System. In *Proceedings of the Eleventh ACM Symposium on Operating System Principles*, November 1987.
- [29] Young, M. Exporting a User Interface to Memory Management from a Communication--Oriented Operating System. Ph.D. dissertation, Carnegie Mellon University, November 1989. CMU-CS-89-202.
- [30] Zajcew, R., Roy, P., Black, D., Peak, C., Guedes, P., Kemp, B., LoVerso, J., Leibensperger, M., Barnett, M., Rabii, F., Netterwala, D. An OSF/1 Unix for Massively Parallel Multicomputers. In *Proceedings of the Winter 1993 USENIX Conference*.



# In-Kernel Servers on Mach 3.0: Implementation and Performance

*Jay Lepreau   Mike Hibler   Bryan Ford   Jeffrey Law*

Center for Software Science  
Department of Computer Science  
University of Utah  
Salt Lake City, UT 84112

E-mail: {lepreau,mike,baford,law}@cs.utah.edu

## Abstract

The advantages in modularity and power of microkernel-based operating systems such as Mach 3.0 are well known. The existing performance problems of these systems, however, are significant. Much of the performance degradation is due to the cost of maintaining separate protection domains, traversing software layers, and using a semantically rich inter-process communication mechanism. An approach that optimizes the common case is to permit merging of protection domains in performance critical applications, while maintaining protection boundaries for debugging or in situations that demand robustness. In our system, client calls to the server are effectively bound either to a simple system call interface, or to a full RPC mechanism, depending on the server's location. The optimization reduces argument copies, as well as work done in the control path to handle complex and infrequently encountered message types. In this paper we present a general method of doing this for Mach 3.0 and the results of applying it to the Mach microkernel and the OSF/1 single server. We describe the necessary modifications to the kernel, the single server, and the RPC stub generator. Semantic equivalence, backwards compatibility, and common source and binary code are preserved. Performance on micro and macro benchmarks is reported, with RPC performance improving by a factor of three, Unix system calls to the server improving between 20% and a factor of two, and 4-13% performance gain on large benchmarks. A breakdown of the times on the RPC path is also presented.<sup>1</sup>

## 1 Introduction

The modularity of microkernel-based operating systems (OS's) provides them well-known benefits. These benefits include improved debugging facilities and software engineering improvements which result from a higher degree of system structuring. The use of separate protection domains to implement systems on top of a microkernel provides robustness; the ability to compose servers to exhibit different OS "personalities" offers flexibility. In addition, microkernels export powerful abstractions which are useful for building distributed systems and multicomputer OS's. However, this modularity and power has come at some cost. Because the modularity is strictly at the task level and tasks communicate via a powerful but "heavy" interprocess communication (IPC) mechanism<sup>2</sup>, all inter-task interactions are expensive. The expensive inter-task operations include the common case of untrusted clients calling trusted servers on the same machine, passing small amounts of data back and forth.

<sup>1</sup>This research was sponsored by the Hewlett-Packard Research Grants Program.

<sup>2</sup>Although tasks can communicate through shared memory, much inter-task interaction is handled via remote procedure calls (RPCs). A few common I/O operations, such as Unix `read`, are often implemented with shared memory and can be reasonably fast.

Although the Mach 3.0[1, 8] microkernel has shown isolated instances of performance comparable to that of macrokernel systems[6], this has not been observed in general. Many software layers and some hardware boundaries must be traversed even for the most trivial interactions between tasks. All messages exercise much of the IPC mechanism, even though common servers, such as the Unix server, use only a small subset of its capabilities. The conclusion is that for inter-task calls, Mach has failed to optimize the common case.

Instrumentation of the Mach IPC path on the Hewlett-Packard PA-RISC has shown that at least on this platform, the traditional “context switch” from one address space to another is only a minor component of same-machine Mach RPC costs. As detailed later, most of the cost is in kernel entry and exit code, data copies in the MIG-generated stub routines, and in port shuffling and related code. We have largely eliminated the last two costs in the common case.

To optimize the common case, we have constructed a separately linked OSF/1 server image (referred to as the *in-kernel server* or *INKS*) that can be loaded into the kernel protection domain at run time. We dynamically replace heavyweight IPC-based communication mechanisms with lighter-weight trap and procedure call mechanisms where possible. We have extended the RPC stub generator to produce special stubs which enact these new mechanisms. Since the change of communication mechanism is essentially a change in procedure binding, the presence or absence of these mechanisms is transparent to clients.

In the rest of this paper we discuss the goals, constraints, and design in Section 2, give implementation details in Section 3, and give status and performance results in Section 4, including breakdown of the RPC path, microbenchmarks, and macrobenchmarks. Finally, we examine related work, discuss planned and possible future work, and give our conclusions.

## 2 Design

### 2.1 Goals

Our goals were five-fold. Most importantly, we wanted to improve the *performance* of systems based on Mach — both those implementing Unix and more specialized systems, such as network protocol servers. Nearly as important, we sought to be as *compatible* as possible, as elaborated below. We wanted to make the required system transformations *acceptable*, by imposing minimal constraints on Mach program structure. We wanted to *explore the issues* involved, and finally, we wanted a sample system as a *target* for a more dynamic binding mechanism, based on an object server we have implemented[9, 10].

### 2.2 Compatibility Constraints

To maintain compatibility, we imposed a number of constraints on the design:

- The source for in- and out-of-kernel servers must be identical, with little use of conditional compilation.
- All changes to the microkernel must be backwards compatible. That is, old clients and servers that use normal RPC's must still run.
- New clients must be able to function with old servers. With a small amount of effort, new clients and servers should function with old microkernels. Note that for Unix programs, compatibility is not an issue, because the “emulator” encapsulates all of the interface to Mach, and will normally match the installed server.

Beyond our primary goals, we have achieved additional compatibility:

- Fully linked binaries of in- and out-of-kernel servers are identical.

- “Conformant” servers, which are defined to be those which don’t directly call `mach_msg`, but only use MIG-generated stubs, currently require only a few lines of “boilerplate” source code in order to take full advantage of our system. (These could be eliminated by modest further changes to MIG, or to pieces of the Mach library.) “Non-conformant” servers run correctly, but with only a slight speed increase over out-of-kernel servers.
- The C threads library required no changes (with the exception of one bug fix).
- Changes to the OSF/1 server and emulator were modest, and predominantly stemmed from its code heritage as a macrokernel.

## 2.3 Design Overview

First we will outline the changes to the kernel and to the RPC stub generator. In principle, these are the only changes needed to allow “conformant” servers to run inside the kernel, with full performance. Then we will trace a client-to-server call, which will demonstrate the overall design. Later sections will describe the special problems that the OSF/1 server and emulator presented.

When loaded into the kernel, a server continues to exist as a distinct task but with two unique attributes: it shares the kernel address map (i.e., lives in the kernel address space) and its threads run in the same privileged mode as kernel threads. Two changes were made to the kernel in support of this. First, a mechanism was added to load a server into the kernel address space and instantiate it as a separate task. Second, existing support for kernel tasks had to be extended, both to support preemption of its threads and to allow them to perform system calls. In addition, to support user to server system calls, we added kernel infrastructure for server stack management and call “registration.”

The other key component in our design is the RPC stub generator, which for typical Mach clients/server applications, is the Mach Interface Generator (MIG). We have produced an extended version of MIG, called KMIG, which includes facilities to supplement remote procedure calls by generating special traps through a dispatch table. The special stubs produced by KMIG allow user-mode client programs to use more efficient control paths when making calls to servers loaded into the kernel protection domain. This mechanism could also be made available to those servers, when making calls to other servers in the same domain.

If a server is linked with the corresponding server-side KMIG-generated stub routines, when it is started it can optionally be loaded into the kernel. Once loaded, the server’s first action is to register with the kernel the RPC procedure ID ranges it supports as client system calls, along with a vector of pointers to its server-side stubs. These tables are automatically generated by KMIG.

The KMIG-generated client stubs are essentially conventional system call traps. The system call numbers are used by the kernel to index into the appropriate server registered table.

When a client executes a server-provided system call, the kernel observes that it is out of the normal range, finds the server’s receive port from the client’s send right name, and looks up the registered trap table for that server. If such a table exists, and the kernel finds the vector index within it, it sets up some server context (stack, global base pointers, and task identity) and dispatches to the associated server-side stub. Otherwise it returns an error code which tells the client to build a message and send it normally.

The server-side stub copies in to kernel space any necessary arguments and calls the actual work function. Upon return from the work function, the stub copies out any returned values and returns to its caller. The kernel then restores state, and returns to user mode. The client-side stub checks the return value for the special error code, and sends a real message if so. Otherwise it just passes back the return code, since kernel code has done all the parameter handling.

In the design outlined here for traps to in-kernel servers, Mach ports are not used as communication channels, but are still used for addressing and protection. A client task must own a send right for an in-kernel server in order for it to be able to make trap calls to that server. Also, in-

stead of being global, trap tables for in-kernel servers are attached to the receive ports owned by that server. In a multiserver environment, this allows multiple in-kernel servers to field overlapping sets of call (trap) numbers, using message ports to distinguish between them. This contrasts with conventional (macrokernel-style) trap semantics, in which there is only one global trap table shared by all user-mode processes.

Note that in this model, user threads are performing server functions for themselves rather than having these functions performed on their behalf by a server-provided thread. If effect, the user thread has “migrated” from its task into the server task. This has numerous implications that are briefly discussed later, and in more detail in [5].

### 3 Implementation

To date we have experimented with the OSF/1 server and a number of special purpose micro-servers. The following sections detail the changes necessary to make these servers run both in and out of the kernel and also discusses the shortcomings of various aspects of the implementation.

#### 3.1 Server Loading

An in-kernel server is currently loaded at boot time using a slightly modified version of the existing bootstrap routines. The server is assumed to have been statically linked at an address that doesn't conflict with any allocated kernel space. During an interactive boot, in addition to prompting for a server name, the bootstrap code now asks whether the selected server should be loaded in or out of kernel.

During a normal boot, the kernel creates a separate kernel bootstrap task (which shares the address space of the kernel), gives it send rights for the master host and device ports and creates a thread to run in it. Once running, this thread uses the standard external kernel interfaces to create a user task and thread, load its address space with the server image and send it the host and device ports. This kernel thread then becomes the initial thread for the default pager.

For the in-kernel case, the kernel creates a second kernel task and thread in addition to the previously described bootstrap task. This second task also shares the kernel address space and will become the server. The bootstrap task is given send rights to the task and thread kernel ports for the eventual server. When the bootstrap thread begins execution it skips creation of a user task and thread and uses the kernel created equivalents instead. From this point on, the bootstrap procedure continues as in the out-of-kernel case. It should be noted that even though the server is now loaded in the kernel's address space, it is still pageable.

#### 3.2 Server/Kernel Interaction

At the current time, we do not attempt to short-circuit any server to kernel interactions. A server running in-kernel still has the same kernel interface as it would running out of kernel. Though considerably simplifying our job, there were still some problems.

One issue that arose is the semantic difference between kernel and user tasks. In standard Mach 3.0, kernel tasks are created with a special kernel function which returns an internal structure identifier instead of a port name. In order to avoid changes to the server code, this name had to be “externalized” and the in-kernel server's task had to be made to work like a normal task port name. Since the kernel only creates other kernel tasks at boot time, the amount of code affected was small and this was not a major problem.

In addition, kernel tasks are directly linked to the kernel address map, instead of having a private map referencing the same memory objects, as is the case when two user tasks share memory. If a server performs a “random” `vm_deallocate` it can actually unmap part of the kernel. Though this

seems unlikely in a well-behaved server, it happens in the Unix server.<sup>3</sup> Our solution was to fix the offending code. A related problem is that the kernel address map contains a particular construct, a “sub\_map,” that does not appear in user task maps and which some of the virtual memory routines cannot handle. One of these routines is `vm_map_fork`, the address map duplication primitive called when a newly created task inherits memory from the parent. Hence, if a server attempts to create a new task inheriting memory from the server, a kernel panic results. This again happens in the Unix server where it “forks” a child task for the `init` process. In this case, since the child task never actually references the inherited memory (the address space is almost immediately deallocated) it was easy to work around.

Though not strictly necessary for correct behavior of the servers involved, server thread pre-emptibility was implemented. This is a problem unique to in-kernel servers since server threads are running as kernel threads and Mach kernel threads are not preemptible. This creates a potential mismatch with server threads which might assume pre-emptibility and that, in turn, could lead to latency problems in the kernel.<sup>4</sup> To be safe, kernel threads running server code are now subject to rescheduling at the “traditional” points (leaving traps, interrupts, system calls). There were two primary difficulties in doing this. One was differentiating a kernel thread running server code from a normal kernel thread. The other was in implementing a new mechanism to force a rescheduling trap since the conventional technique is to use an AST that is triggered upon return to user mode. Both were solved in highly machine-dependent, and temporary, ways.

A special, lighter-weight system call interface was introduced to handle Mach system calls from the server to the kernel. Since it is a kernel-to-kernel transition, less state saving and setup were needed and there was no need to check for emulated (i.e., server-handled) system calls. Measurement of a “do nothing” kernel call (`task_terminate` with a NULL argument) shows that this modified path cuts over 30% from the overhead of the standard system call interface.

Saving server state when entering a system call presented a minor problem. In the normal server/kernel model, a user thread (application or OS server) has two sets of saved state. Whenever it traps into the kernel (system call or interruption) some or all register state is saved in the process control block (PCB) in a “saved state” structure. When it is context switched (always from within the kernel) the volatile kernel state for the thread<sup>5</sup> is saved at the base of the active kernel stack. In the INKS world, an additional level of state saving is necessary since a user thread may first trap into the kernel to perform a server function and then, in the context of the server, trap into the kernel again to execute a Mach service. The latter cannot use the standard saved state structure since it contains state from the initial user to server transition. In our implementation we allocate another saved state structure in the PCB for use during server to kernel transitions.

Kernel stacks were another source of problems. In principle, during a Mach system call an in-kernel server thread could continue to run on its own stack instead of switching to a kernel allocated stack. This would eliminate the need to copy arguments, switch stacks, and in general, would simplify the system call path. The problem, however, is that the special optimized paths through the existing kernel assume they have complete control of the stack the thread is running on (not an unreasonable assumption in a “pure” kernel environment). In particular, continuations are a problem because their effect is to throw away the contents of the kernel stack. Since there is now pre-system call server state on the “kernel” stack we could not allow this.<sup>6</sup>

Though general disabling or modification of the continuation system and the related stack handoff code is a possibility, it would have required code changes we judged too extensive at this time. Therefore, in order to preserve the semantics of kernel stacks, we chose to have the server run off of its own stack and switch to a kernel stack on system call entry. This means we now need to copy

<sup>3</sup>On booting, the server deallocates page zero of “its” address space but since the server is no longer loaded at zero, this deallocates the first page of the kernel.

<sup>4</sup>As an extreme example, a naive server running in-kernel might loop on a spin lock causing the kernel to deadlock.

<sup>5</sup>There may be no volatile state if a continuation was specified at context switch time.

<sup>6</sup>We implemented a prototype of this and obtained about a 35% speedup of the server-kernel system call path. As a workaround in this prototype, the affected continuations were identified and disabled. (This allowed further streamlining of the system call path as we no longer needed to save callee-save registers for `thread_syscall_return`.)

arguments, save registers for continuations, and do some additional stack management. Essentially, we were forced to retain much of the original system call path, even though no protection boundary is being crossed. We still save some overhead, but the principal performance benefits arise from coupling this with changes to the RPC stubs.

### 3.3 User/Server Interaction

KMIG treats specially routines with “normal” arguments and message options (in particular, no port rights or out-of-line memory regions). It generates client stubs which are merely system calls whose number corresponds to the `msg_id` number (RPC subsystem base + procedure ID number). The user stack already contains the arguments, including the server’s port name, just as the user supplied them in the C-style function call, so no argument copying is needed.

When a trap is made to the kernel, it first determines whether it is a “normal” trap (to Mach primitives or Unix emulation) or a trap to an in-kernel server. This distinction is easy to make because normal traps use small positive or negative numbers, while INKS traps, whose trap numbers correspond to `msg_id` numbers, are large positive numbers.

When the kernel determines that the trap is to an in-kernel server, instead of looking up the trap in a global table as normal traps are handled, it first determines *which* server is being called. It retrieves the server’s port name, which the user supplied as the first parameter in the call<sup>7</sup>, and decodes it in the context of the client’s IPC space to find the destination port and the task in which it resides (the server task). From that task, it extracts the trap table previously registered by the in-kernel server and looks up the stub entrypoint based on the `msg_id` number originally supplied as the trap number.

Before calling the server-side stub, the kernel switches from its stack to a new server-provided stack. Note that this violates our “ideal” in that it requires server changes to make stacks available for trapping clients. There are three major reasons why we must do this. First is the continuation/stack-handoff issue discussed earlier: remaining on a kernel stack while running in the server limits optimizations that the kernel can do when the server later makes a Mach system call. A second, more serious problem is that server routines may make assumptions about the layout of the stack they are running on. For example, the OSF/1 server allocates a thread specific data structure (the `uthread` structure) on the stack of every service thread.<sup>8</sup> All references to this structure are made via a constant offset from the base of the current stack, which is assumed to be a fixed size Cthread stack. Hence, not only does the server have knowledge of the layout of the stack but it also make assumptions about its size and alignment. Also, since OSF/1 service threads are Cthreads, there is additional Cthread-specific state on the stack as well. Finally, kernel stacks are relatively small and wired-down in the kernel. The former means that they may not be large enough for complex or deeply nested server calls, while the latter could lead to excessive wiring of physical memory.

After the kernel has switched to a service stack, the system call arguments are copied from the user’s address space to the server’s. (Note that these are normal, short, call-by-value arguments.) The kernel’s final action before the KMIG-generated server stub is called is to change the current thread’s “task identity” to that of the server task. Since our desire is to run server code largely unchanged, we must preserve the notion that it is running in a separate task. Hence, server code which performs operations on “`mach_task_self`” should affect the in-kernel server task and not the user task, even though it is running from a user thread. Our current approach to solving this is ad hoc. When a user thread traps into the server, we change its containing-task pointer to that of the server task. We do not currently remove the thread from the task’s thread list and insert it into the server task’s.

Once the server stub receives control, it allocates local variables for the storage of pass-by-

<sup>7</sup>MIG allows the request port name to be in other positions in the argument list, but we do not know of any case in which this feature is actually used. If this feature were to be used, KMIG would simply generate a standard message stub for that procedure.

<sup>8</sup>This structure contains process information that is only valid while in a system call, so the service thread stack is a convenient place for it.



reference or Out arguments, copies pass-by-reference arguments from user space with the kernel's standard copyin procedure, and calls the server work function. Upon return from the work function, the server copies any Out or InOut arguments back into user space via copyout and returns to the kernel. The locations of the kernel's copyin and copyout procedures were returned as part of the server's trap table registration.

Finally, the kernel resets the thread's task pointer to its pre-call value, switches back to the original stack and exits the client's trap.

The Unix server required two deviations from the standard procedure. The first deals with copying arguments in to and out of the kernel address space and is explained below. The other is a violation of our "conformant" constraint that servers not directly call mach\_msg. For brevity, the Unix server and emulator implement a number of services with a "hand-rolled" generic RPC stub, without using MIG. We had to similarly hand-implement a generic INKS trap to handle these system calls.

### 3.4 Issues

#### 3.4.1 Thread Model

One important implementation issue we had to confront is the thread semantics employed during traps to in-kernel servers. Should the client's thread "move" into the server task for the duration of the RPC, or should the kernel switch to a "real" thread created and owned by the server? The latter adheres to existing Mach semantics, and therefore would be more straightforward. However, for our initial implementation we chose the former option, for several reasons: it offers the greatest potential for performance improvement from INKS, it tests the feasibility of such a stretching of Mach semantics, it offers the flexibility to support other specialized communication mechanisms, and we did not realize the extent to which the Unix server relied on "specializing" its service stacks.

In [5] we discuss in detail the issues related to this decision, so only a brief summary is presented here. A thread switching model would have resulted in a cleaner, safer<sup>9</sup> implementation and would have fit better with Mach's current thread semantics. It would probably be slower than a migrating threads implementation, but perhaps not significantly overall, especially for a single server system. In the long term, however, we believe that thread-switching is only a temporary way to get around a problem in Mach. If a general-purpose migrating threads model such as that exploited in LRPC[3] is introduced to Mach in the future, this approach to in-kernel server traps will become both faster and cleaner.

#### 3.4.2 Server Task

Many of the complexities and shortcomings of the thread migration model could be avoided and the benefits retained by altogether eliminating the notion of a separate server task. By viewing the server code as down-loadable kernel code running as part of the "kernel task" we no longer have to worry about thread migration or service threads. User threads trapping into the server are just trapping into an extended kernel, and the user thread is just "running in kernel mode". The semantics of thread operations applied to user threads running in the kernel are well-defined, so threads running in server code introduce no new problems. Coupling this with short-circuiting server/kernel calls, we have essentially re-created a monolithic kernel.

There are drawbacks, however. One is the kernel/server stack issue. User threads entering the kernel will still need to switch to a service stack or "prepare" the kernel stack so it appears as the server expects. For the latter, the size and non-pageable nature of kernel stacks would again be a concern. In either case, the kernel would have to be modified to avoid using the continuation and stack handoff optimizations or those optimizations would have to be modified extensively. This

<sup>9</sup>Currently, when a thread migrates into a server, it can still be manipulated by any other thread that has access to its thread control port.

approach also does not address the preemptibility problem. Since server threads would now be true kernel threads, they would no longer be preemptible, leading to all the attendant latency problems. It is also not clear how much an existing server would need to be changed to fit this model. While many of the kernel to server interface issues can be addressed by KMIG or more sophisticated linking technology, much depends on what assumptions a server makes about its address space and other resources.

### 3.4.3 Trap Tables

In our current implementation of INKS, the trap tables for an in-kernel server are attached to its task, instead of individually to each of its service ports. While this simplifies the implementation and works well in the case of the Unix server, it may have to be changed in the future. The problem is that, because one receive port in the server's task is not distinguished from another, *any* send right whose receive right is in the in-kernel server's task may be used by other tasks to make INKS traps into the server, even if that port is not one that would respond to corresponding messages. This is not a problem in the Unix server, because there is only one "type" of service port that clients know about: the BSD request port. However, there could be a problem in a server that exports multiple "types" of send rights, each responding to a different set of request messages.

There are two apparent ways of solving this problem. First, if the implementation of INKS is left the same, server work functions could simply check the receive port that corresponds to the send right the client used to make the trap, ensuring that the call being made is in fact valid for the given port. However, a better and more "correct" way to do it would be to simply change INKS so that trap tables are associated with individual receive ports in the server, instead of the server's task. This would be somewhat more difficult to implement than the current strategy, but we foresee no major problems.

### 3.4.4 Copyin and Copyout

As described earlier, KMIG generates calls to the kernel copyin and copyout routines to deal with copying arguments and results across the user/kernel boundary. This led to an unexpected inefficiency in the Unix server. Since it is derived from a monolithic kernel, most of the system call service routines already do their own copyin and copyout. Hence we were doing two copies for every data item, in or out.

To solve this, KMIG was modified to optionally omit generation of all copyin and most copyout calls. The only KMIG-generated copyouts which remain are for integer Out parameters which are traditionally return values.

Unfortunately, eliminating KMIG-generated copies uncovered a few places in the Unix server where arguments were not being handled consistently. For example, the `bsd_execve` routine (not inherited from the monolithic kernel) dereferences some of its out pointer parameters assuming they will be accessible in the server's address space. However, the KMIG stub no longer allocates a local array for these parameters, expecting the service routine to do its own copyout. We view this as a server problem and fixed the offending routines.

In total, the changes to perform this copy optimization in the OSF/1 server were limited to 3 files and 7 functions. Whether the KMIG option to suppress copies is generally useful or peculiar to the Unix server remains to be seen.

### 3.4.5 Miscellaneous

Finally, there were a number of other issues that came up and were not resolved to our satisfaction:

- The problem of inheriting an in-kernel server's address space (i.e., the kernel address space) on task creation needs to be addressed. Although the problem was easily avoided in the case of

the Unix server, in general, other servers may want to export portions of their address space.

- A related creation issue is whether a kernel task should be able to spawn only kernel tasks, just user tasks, or both. Currently, `task_create` will always create a user task, which is sufficient for the Unix server.
- There is a thread analog to the “task identity” problem discussed earlier. Just as server code may make the assumption that `mach_task_self` does not refer to the user task, it may also assume that `mach_thread_self` does not refer to the user thread, or, more importantly, that the currently running thread is not the user thread. An example of this in the OSF/1 server is the `exit` system call which attempted to terminate the currently active user task and threads, one of which is running the exit code.
- The existing fixed partitioning of a thread’s saved state may need reworking. For the single in-kernel server experiments we were doing, simply adding another “saved state” structure to the PCB for server/kernel transitions was sufficient. However, introducing multiple servers with server to server interactions will add arbitrary levels of state saving. Presumably such inter-server calls can be handled by saving the necessary old server state onto the new server’s stack thereby requiring no additional static saved state areas.
- If a server is overloaded, with messages built up in its port queue, clients invoking it by the system call interface will get preferential treatment, since the direct invocation bypasses the port’s queue. Handling this is straightforward: if the kernel finds the server’s receive port has a non-empty queue, it returns `MACH_SEND_INTERRUPTED`, signaling the client stub to send a real message.
- To this point we have concentrated solely on increasing execution speed and have not considered the effect of in-kernel servers on memory usage or the effect of its potentially increased memory use on overall system performance. Since in-kernel servers are pageable their mere existence in the kernel does not imply increased physical memory usage. However, it does increase virtual memory use within the kernel address space.<sup>10</sup> The current implementation, with its private service stack pool for trapping clients, certainly increases memory usage.

## 4 Results

### 4.1 Status

The identical OSF/1 single server binary runs multiuser, in or out of the kernel. With trapping clients there are occasional robustness problems, but the system typically stays up for hours under benchmark loads (SPEC SDM, Andrew, kernel builds). Some types of signals crash the system, but we have not examined this yet.

### 4.2 Experimental Environment

All timings were collected on a single HP9000/730 with 32 MB RAM and one 425 MB SCSI-2 disk, on an isolated ethernet segment. The HP730 has a 67 Mhz PA-RISC 1.1 processor, 128K offchip Icache, 256K offchip Dcache, 96 entry ITLB, and 96 entry DTLB, with a pagesize of 4KB. The caches are virtually addressed with a cache miss cost of about 14 cycles. RPC test times were collected by reading the PA’s clock register, CR16, which increments every cycle, and can be read in user mode. This was done with an inline function and `asm`. Other times are obtained from the Unix server.

<sup>10</sup>It has been suggested that the presense of the server in the kernel’s address map may affect the performance of map entry lookups at fault time, especially on server faults where we might potentially need to scan past all the kernel entries just to reach the server’s “neighborhood”.

Table 1: Non-trapping RPC Test Program Breakdown

RPC Stage	Description	Out of Kernel Cycles	In Kernel Cycles
1	Procedure call to MIG stub	24	24
2	Stub operations before mach_msg_trap call	199	199
3	Call to mach_msg_trap, trap into kernel	350	369
4	mach_msg_trap operations before message copyin	18	18
5	Copyin message to kernel address space	205	205
6	"Red tape"	159	159
7	Context switch from client task to server task	178	178
8	"Red tape"	122	121
9	Copyout message to server, trap return, MIG demux	443	404
10	Server MIG stub processing before work function	19	19
11	Null work function (just returns)	17	17
12	Server MIG stub processing after work function	172	172
13	Server call to mach_msg_trap, trap into kernel	378	304
14	mach_msg_trap operations before message copyin	18	18
15	Copyin reply message to kernel address space	211	212
16	"Red tape"	144	144
17	Context switch from server task to client task	178	179
18	"Red tape"	90	90
19	Copyout reply message to client, trap return	364	364
20	Client stub operations after mach_msg_trap call	172	173
21	Procedure return from MIG stub to client	17	17
	Total	3478	3386

The operating system software is the Mach 3.0 kernel, version NMK13, and the OSF/1 single server, version 1.0.4b1 (derived from OSF/1 1.0.4). This is the port done by Utah, and does not yet include the virtual cache improvements from CMU[13]. Little of the port has been optimized, and in particular, the existing implementation of the system call/context switch path is not at all optimal. This likely affects the breakdown of RPC costs, but probably has only a marginal affect on the overall results. The compiler is GCC 2.3.3.u3, (a Utah distribution of 2.3.3 with additional optimizations for the PA). All RPC timings were made with only the micro "RPC server" running, and all Unix benchmark measurements were made with the system in single-user mode. Each Unix run was repeated at least three times. The "RPC server" timings showed essentially no variance, with only 1-3 cycle differences out of thousands.

### 4.3 RPC Breakdown and Analysis

In Table 1 we present a breakdown of an RPC to an out-of-kernel server, containing one InOut parameter of 32 words, which is used to store timestamps at successive stages of the message path. This test does not, and is not meant to, completely represent "typical" RPCs. The most common RPCs pass only a few (three to ten) discrete integer parameters, most of which are In or Out; InOut arguments are less common. However, the purpose of this test is not to measure the total performance of an RPC, but to determine comparatively how much time the various parts of the RPC path take. The most important requirement is that all measured code remain on the "optimized path" within mach\_msg\_trap, and our test fulfills this requirement. We now briefly describe the

Table 3: RPC test results: cycles (ratio to trapping INKS)

Test	Configuration		
	Unoptimized	INKS Message	INKS Trap
Null RPC	2312 (3.5)	2220 (3.4)	656
64 In	2555 (3.1)	2462 (3.0)	830
1K In	5843 (3.3)	5890 (3.3)	1791
1K Out	8366 (3.0)	7512 (2.7)	2741
128 InOut	3385 (3.0)	3291 (2.9)	1122

Trap stages 2–6 contain most of the processing done by the microkernel for the INKS trap. In stage 3, the mapping from the client's port name to the kernel's `ipc_port` pointer is done, as well as the lookup of the requested trap in the server's INKS trap table. Stages 4 and 5 handle the selection, setup, and activation of a new service stack for use by the server's work function. Stage 5 also includes changing the task context and copying the client's basic (call-by-value) parameters to the new server stack.

Trap stages 7–9 correspond closely to RPC stages 10–12, except that *all* required argument copying is done in the server stub. The large times for stages 7 and 9 stem from the explicit copying of the 32-word argument into the server's address space and, later, back out to the client's. This is the *only* argument copying that is done in the trapping case, unlike the RPC case where the arguments are copied in the client stub and in the kernel as well. However, note that while the large time spent in RPC stage 12 is an artifact of the test implementation as discussed previously, the time spent in the corresponding trap stage 9 is *not* such an artifact—this copy would in fact have to be done for *Out* arguments as well as *InOut* arguments, which are quite common.

For comparison, a simple procedure call on the same machine typically takes 15 to 25 cycles.

Based on these tables, an approximate categorization can be made of the time spent in different types of processing during the RPC path. Argument copying accounts for about 1350 cycles in the RPC case, but only about 450 for the INKS trap. Trap entry and exit times are also significantly better in INKS, from about 1100 cycles down to about 400. Finally, most of the “red tape” involved in passing Mach messages is eliminated.

It is clear that much less argument copying is done in INKS traps than in RPC. Simple stack-based *In* parameters are copied only once (client stack to server stack) instead of four times (client stack to client message, client message to kernel, kernel to server, server message to server stack). Pointer-based *In* parameters are copied once instead of three times (the copy from server message to server stack is not necessary in the RPC case). *Out* parameters are similarly reduced from three copies to one. *InOut* parameters are copied only twice instead of seven times (client to message, message to kernel, kernel to server request message, request message to reply message, reply message to kernel, kernel to client message, reply message to original buffer).

#### 4.4 Micro Benchmark results

Table 3 shows several small RPC benchmarks, run directly on the Mach kernel, with no Unix server running. Five tests were run: a null RPC, a 64 byte *In* parameter, a 1024 byte *In*, a 1024 byte *Out* parameter, and a 128 byte *InOut* parameter (the same test as in Tables 1 and 2 without the instrumentation code). The cycle count for each test under each configuration is shown, along with a ratio to the trapping INKS configuration, in parentheses. The unoptimized message configuration and INKS message configuration typically execute three times as many cycles as the INKS trap configuration. Variance in these tests was almost nonexistent; on rare occasions the results would vary by a single cycle. From the *InOut* results in the three RPC tables, one can measure the overhead

Table 2: Trapping Test Program Breakdown

Trap Stage	RPC Stage	Description	Cycles
1	1	Procedure call to user stub	24
2	3	Trap into kernel, recognize the trap as INKS	213
3		Port, task, and trap entryptpoint lookups	116
4		Find and allocate a service stack	22
5		Switch stacks, change task context	92
6		Call server stub	29
7	10	Server stub processing before work function ( <i>copyin</i> )	232
8	11	Null work function (just returns)	20
9	12	Server stub processing after work function ( <i>copyout</i> )	229
10		Return from server stub to kernel	17
11	19	Deallocate stack and return from trap	163
12	21	Procedure return from user stub to client	17
		Total	1174

contents of the table.

Stages 1–2 and 20–21 occur in the client’s address space. Stage 2 is the marshalling of the request message, and stage 20 is the unmarshalling of the reply message. Note that both of these stages are somewhat higher in our test case than they would be in common RPCs, because of the larger-than-normal *InOut* parameter that must be copied into and out of the message.

Stages 3–9 and 13–19 occur mostly in the kernel’s address space (in particular, in the *mach\_msg\_trap* function). Stages 3–9 are essentially identical in overall function to stages 13–19. (The code path executed in stage 6 is actually different from the one executed in stage 16, but the overall functionality and execution time for each is very close.)

“Red tape” (stages 6, 8, 16, and 18) is code in *mach\_msg\_trap* that performs administrative tasks necessary to the passing of simple messages: port name lookups, generating reply rights (in the case of request messages), consuming request or reply rights, locking and unlocking kernel data structures, as well as numerous tests for exceptional situations that would require control to be transferred off of the optimized path.

Stages 10–12 happen in the server side. Stage 12, like stages 2 and 20, is much larger than it would be for “real” RPCs. In the server, the received request message is stored in a message buffer separate from the reply message to be built and returned. While this does not matter for *In* or *Out* arguments, which are the most common, *InOut* arguments must be copied by the server stub from the request message to the reply message. Most of the time spent in stage 12 is in this copy, and if the argument was not *InOut*, the stage 12 time could be expected to be similar to the stage 10 time.

Table 2 gives the figures for the same test using the in-kernel server trap mechanism. Again, a single 32-word *InOut* argument is passed. The “Trap Stage” column independently enumerates the stages of an INKS trap, while the “RPC Stage” column lists the stage numbers from Table 1 that correspond to the stages in Table 2, where appropriate.

In the trapping case, the time spent in the client-side stub (trap stages 1 and 12) is very small because the client stub is a trivial assembly language fragment that simply loads the trap number into a register and makes the trap.

Table 4: Syscall test results: time in microseconds (ratio to trapping INKS)

Test	Configuration		
	Unoptimized	INKS Message	INKS Trap
nop (emulator)	11.3 (0.95)	11.3 (0.95)	11.9
nop (server)	92.7 (2.08)	86.7 (1.94)	44.6
getpid	11.9 (1.00)	11.9 (1.00)	11.9
getgid	92.6 (2.18)	85.9 (2.02)	42.5
setrlimit	114.2 (1.94)	105.6 (1.80)	58.7
4096 byte-read	813.8 (1.19)	802.1 (1.18)	682.4
64 byte-read	310.0 (1.70)	301.2 (1.64)	183.5
4096 byte-write	489.7 (1.23)	483.2 (1.22)	396.5
64 byte-write	179.4 (1.98)	172.7 (1.91)	90.5
stat	280.5 (1.55)	273.2 (1.51)	180.9
create	535.6 (1.69)	518.8 (1.64)	317.2
signal	394.1 (1.46)	372.5 (1.38)	269.1

Table 5: Server to kernel syscalls: Time in microseconds (ratio to INKS)

Test	Configuration	
	Unoptimized	INKS
Message Syscall	18.81 (1.13)	16.71
Trapping Syscall	6.86 (1.44)	4.75

imposed by instrumenting the message and trap paths. It is both minimal and consistent: 2.6% (93 and 95 cycles) for the two message paths, and 4.6% (52 cycles) for the trapping path.

As an aside, during this work we discovered a source of huge inefficiency in the standard MIG-generated stubs. To copy an argument “struct” into and out of messages, MIG generated a structure containing a single `char` array, and used structure assignment to effect the copy. Both GCC 1.39 and HP’s C optimizing compiler generate code which copies a byte at a time, even if the array is word-aligned (as it usually is). Moving to GCC 2, or replacing the code with a `bcopy`, improved RPC message performance by a factor of two; before this INKS trapping RPCs attained a factor of six speedup. The lesson is that large potential improvements can lie in mundane and seldom-examined portions of the system.

Table 4 shows timings for a small variety of arbitrarily selected, but hopefully representative, Unix system calls. These were the only system call benchmarks we ran: no result filtering has been done. Times are in microseconds with the time relative to the trapping INKS configuration shown in parentheses. In calls to the Unix server, the trapping INKS configuration improves Unix system call performance by a minimum of 20%, ranging up to a factor of two. “Nop (emulator)” and “getpid” are handled completely within the emulator and therefore should experience no performance improvement under trapping-INKS. Then times on the two message configurations typically varied by 2% to 5%, while the trapping times varied by only 1% to 3%.

As discussed previously, certain aspects of Mach’s implementation prevented us from achieving a large performance gain in calls from an in-kernel server to the microkernel. However, the fact that the server runs in the kernel address space still allowed some minor optimizations to be made along the trap path, resulting in slightly higher performance. Table 5 compares the times for a do-nothing server-to-kernel system call in the normal case (server running in user mode) and the INKS case (server running in kernel mode). Message syscalls account for most of the general-purpose calls in

Table 6: Macro benchmarks: Time in seconds (ratio to trapping-INKS)

Test	Configuration		
	Unoptimized	INKS Message	INKS Trap
Full kernel build	1682 (1.15)		1463
Andrew (filesystem phases)	48.8 (1.04)	49.5 (1.05)	47.0

Table 7: Communication Counts

	From User	
Emulation Traps	82474	34.9%
INKS Traps	48142	20.4%
Mach Syscalls (trap)	2748	1.2%
Mach Syscalls (msg)	16024	6.8%
Total From User	149388	63.2%
	From Server	
Mach Syscalls (trap)	41480	17.5%
Mach Syscalls (msg)	45654	19.3%
Total From Server	87134	36.8%

the Mach interface such as device I/O, while trapping syscalls are used for the very common cases like `vm_allocate`.

#### 4.5 Macro Benchmark Results

We also ran somewhat more realistic Unix workloads, in particular the modified Andrew benchmark[11] and a Mach kernel build. The more realistic kernel build benchmark showed a 13% speedup under trapping INKS, while the Andrew benchmark yielded 4%. Table 6 shows these timing in seconds, with the ratio to trapping INKS in parentheses.

All runs were done with the system in single-user mode. “Andrew” includes only the four filesystem phases of the benchmark. For that test, after the initial run we found results only varied by at most a second. The kernel build compiles the Mach kernel from scratch, with all files local. Six runs of this test were made, with the last five showing run time variation of less than 1%.

On all system configurations, since the first run of any benchmark was significantly slower than succeeding ones, we factored out the first run of any macro benchmark. An anomaly we noted is that trapping INKS usually paid a higher startup penalty than either message case, typically twice as high. We have not yet investigated the cause of this effect, but speculate that once found, fixing it will yield further improvements in performance, for all runs.

We also ran the SPEC SDM SDET benchmark, and obtained about 8% throughput increase for the low-load “1 script” case. This gain rapidly decreased for higher loads, to less than 1%.

Table 7 shows the usage of the different communication mechanisms during one run of the Andrew benchmark. Table 8 breaks down by message type and source, the messages seen during the benchmark. The dominance of “simple” RPCs confirms our decision to concentrate on this type of RPC. However, the significant number of server-originated messages suggests that optimization of the server-kernel path would be worthwhile.

The data in these tables, together with our micro-benchmark numbers, can be used to validate the system time reduction obtained in larger benchmarks.



Table 8: Mach Message Type Breakdown

	From User	
Sync Simple	7810	12.7%
Sync Complex	8214	13.3%
Total From User	16024	26.0%
	From Server	
Sync Simple	22176	35.9%
Sync Complex	4666	7.6%
Async Send	6552	10.6%
Async Recv	12260	19.9%
Total From Server	45654	74.0%

## 4.6 Cache Issues

One can speculate that some of the performance improvement stems from better cache use. However, the expected cache effect is minor and would tend to worsen performance. (Note that the portion of the virtual address used to access the cache on the PA includes the address space identifier.) Moving the server and kernel into the same address space, thus making equal the space identifier portion of the cache index, should tend to worsen cache performance slightly. Linking the server, as we do, at an address that doesn't conflict with the kernel or user processes, should tend to improve cache performance over a server linked at the normal address. However, in both in- and out-of-kernel cases we use the same server, so this is not a factor. In general, we do not expect cache effects to be significant and *consistent* in this experiment. Other studies have shown that random changes can have significant cache effects[2].

## 5 Limitations

A serious limitation that arises from use of this framework is the loss of the protection that separate address spaces provides. Hardware protection is an important mechanism used to provide robustness in the face of unfriendly or malicious programs. Use can only be made of the merging of protection domains in situations where this loss of protection will not introduce new risks or inconveniences. For example, in the case of a Unix server which is a crucial and trusted operating system component, when loss of the server means (as it typically does) loss of all relevant system services, reduction in protection introduces no new limitations from a user's point of view. The difference between the Unix system dying on a fault in kernel space and dying on a fault in user space is moot. If, however, the service is part of a fault-tolerant system and is capable of recovery from faults, or if it is only one component in a system with other important, independent services, the loss of protection introduces real drawbacks to users and should be carefully weighed.

A limitation of the current implementation is that transferring port rights or out-of-line memory forces a normal message. This is not necessarily a drawback, for if the operation is rare enough, we increase our overall performance by simplifying the code for the common case. Most servers, including the Unix server, do not transfer port rights between themselves and clients or other servers. The vnode pager in the Unix server does issue substantial numbers of Mach kernel calls to do port manipulation, but short-circuiting the server-kernel path is a prerequisite optimizing these calls. Further measurement of the argument type and message frequencies is needed to determine whether is is worthwhile to sort-circuit messages for these arguments.

## 6 Related Work

The most similar work to ours is that done in the Chorus[12, 6] microkernel/multi-server Unix system. Chorus “supervisor actors” correspond to our in-kernel servers, but there are significant differences. Chorus adopts the “migrating-thread” model, with a single kernel stack which remains with the thread. This is easy because continuations do not exist and existing servers do not make assumptions about the intimate details of their stacks (they use native kernel threads). All potential optimizations are performed: user-server, user-kernel, server-server, and server-kernel interactions. The linkage is typically through traps, although procedure call linkage is also used. Supervisor actors are linked separately from the kernel and each other, as we do. In principle, supervisor actors can be loaded and unloaded dynamically, although some violate the rules for this. The client interface to the local Unix “process manager” is *always* through traps; local messages “cannot happen.” Thus there is no check for a non-0 length message queue. Asynchronous messages are not optimized. Chorus’s lack of an RPC stub generator contributes to a relative lack of flexibility and transparency compared to our scheme. The name space (service id’s) of multiple servers cannot overlap in Chorus.

## 7 Future Work

There are a number of ways to continue this work. A useful short term effort is to do the same experiment on another hardware architecture, in order to help factor out machine-dependent effects. We feel that the most important task is to concentrate on *measurements*, so informed decisions can be made between the various implementation strategies. Further investigation of the “ramp up” and scaling problems will likely reveal bottlenecks. Completing the support for server-server interactions would allow us quantify the improvement in a multi-server environment where the speedup should be substantially greater, due to the increased numbers of inter-domain transfers[7].

Optimizing server-kernel interactions should be examined, as it could potentially result in up to two orders of magnitude reduction in overhead (message vs. procedure call). If this optimization were done, note that it generalizes the existing mechanism supporting alternate system call paths to a few Mach kernel calls. This would remove it from the realm of special-purpose “hacks” and make it a universally applied optimization.

In the longer view, we plan to use the powerful linking capabilities of the OMOS meta-object/object server[9, 10] to bind more flexibly and dynamically. OMOS can hook pre-existing programs and modules together in a structured and programmable way. We will move toward making OMOS capable of dynamically binding arbitrary modules with arbitrary interfaces (C-style function calls, C++ method invocations) communicating across arbitrary channels. In this role OMOS is analogous to the nameserver and binding server in a traditional RPC system, but is capable of much tighter and more flexible binding.

## 8 Conclusion

The client-server Mach IPC path can be heavily and transparently optimized in the common case of a trusted Unix server, with little impact on server code. Major improvements in IPC and Unix system call performance result, with significant improvements on Unix benchmarks. The Mach thread abstraction has the potential to evolve to a migrating-threads model.

## Acknowledgements

We are grateful to Douglas Orr for enlightening discussion and review of earlier drafts, Allan Bricker for patient explanation of the Chorus mechanism, and Michael Condict for helpful discussion of the OSF/1 server and INKS issues. We also thank Robert Kessler for reviewing earlier drafts, Leigh Stoller for useful suggestions, Peter Hooenboom for extensive time in table formatting, and

John Bonn for help running benchmarks. In addition, we thank Hewlett-Packard for register CR16 which makes precise low-overhead timing easy.

## References

- [1] Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young. Mach: A new kernel foundation for UNIX development. In *Proceedings of the Summer 1986 USENIX Conference*, pages 93–112, Atlanta, GA, June 9–13, 1986. Usenix Association.
- [2] B. Bershad, R. Draves, and A. Forin. Using microbenchmarks to evaluate system performance. Technical report, Carnegie Mellon University, November 1991.
- [3] Brian N. Bershad, Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy. Lightweight remote procedure call. *ACM Transactions on Computer Systems*, 8(1):37–55, February 1990.
- [4] Bryan Ford, Mike Hibler, and Jay Lepreau. Extending the Mach 3.0 thread model. Technical Report UUCS-93-012, University of Utah Computer Science Department, April 1993.
- [5] D. Golub, R. Dean, A. Forin, and R. Rashid. Unix as an application program. In *Proceedings of the Summer 1990 USENIX Conference*, pages 87–96, Anaheim, California, June 11–15, 1990. Usenix Association.
- [6] M. Guillemont, J. Lipkis, D. Orr, and M. Rozier. A second-generation micro-kernel based unix; lessons in performance and compatibility. In *Proceedings of the Winter 1991 USENIX Conference*, pages 13–22, Dallas, TX, Winter 1991.
- [7] Daniel P. Julin, Jonathan J. Chew, J. Mark Stevenson, Paulo Guedes, Paul Neves, and Paul Roy. Generalized emulation services for Mach 3.0 — overview, experiences and current status. In *Proc. of the Second Usenix Mach Symposium*, pages 13–26, 1991.
- [8] Open Systems Foundation and Carnegie Mellon University., Cambridge MA. *MACH 3 Kernel Interface*, 1992.
- [9] Douglas B. Orr and Robert W. Mecklenburg. OMOS — an object server for program execution. In *Proc. Second International Workshop on Object Orientation in Operating Systems*, pages 200–209, Paris, France, September 1992. IEEE Computer Society.
- [10] Douglas B. Orr, Robert W. Mecklenburg, Peter J. Hoogenboom, and Jay Lepreau. Dynamic program monitoring and transformation using the OMOS object server. In *Proceedings of the 26th Hawaii International Conference on System Sciences*, pages 232–241, January 1993.
- [11] John Ousterhout. Why aren't operating systems getting faster as fast as hardware? In *USENIX Conference Proceedings*, pages 247–256, Anaheim, CA, Summer 1990. Usenix.
- [12] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, S. Langlois, P. Léonard, and W. Neuhauser. Chorus distributed operating systems. *Computing Systems*, 1(4):305–367, 1988.
- [13] Bob Wheeler and Brian N. Bershad. Consistency management for virtually indexed caches. In *Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 124–136, October 1992.



# Redirecting System Calls in Mach 3.0

## An alternative to the emulator

*Simon Patience*

Open Software Foundation  
Research Institute  
2, Avenue de Vignate  
38610 Gières  
France  
Tel: (+33) 76 63 48 72  
E-mail: sp@gr.osf.org

### Abstract:

In order to implement system calls, current Mach 3 based operating system personality servers all make use of Mach's system call redirection mechanism together with an emulator co-resident in the applications address space. Experience with OSF/1 MK has shown that this is problematic for OSF/1 system call semantics, system security and integrity. This paper looks at what these problems are and why they occur. It also describes new kernel mechanisms to implement OSF/1 system calls which address these problems and our experiences implementing them. Finally the benefits and shortfalls of the new mechanisms are discussed both in relation to conventional computer architectures and also with respect to new architectures, such as multi-computers and clusters.

## 1. Introduction

For the past two years the OSF Research Institute at Grenoble, France, has been developing a version of OSF/1 based on Mach 3. The architecture for this system was basically copied, for the most part, from the BSD server from CMU. The model used was that the application programming interfaces were implemented in a single server executing in user space. Access to the server from the application was via an emulator that was resident in a fixed place in the address space of every process. As time passed it was found that the advantages of the emulator also brought many disadvantages, some performance, some semantic and some maintenance. It became clear that, for implementing OSF/1 semantics, the emulator was not the best solution. During the past 9

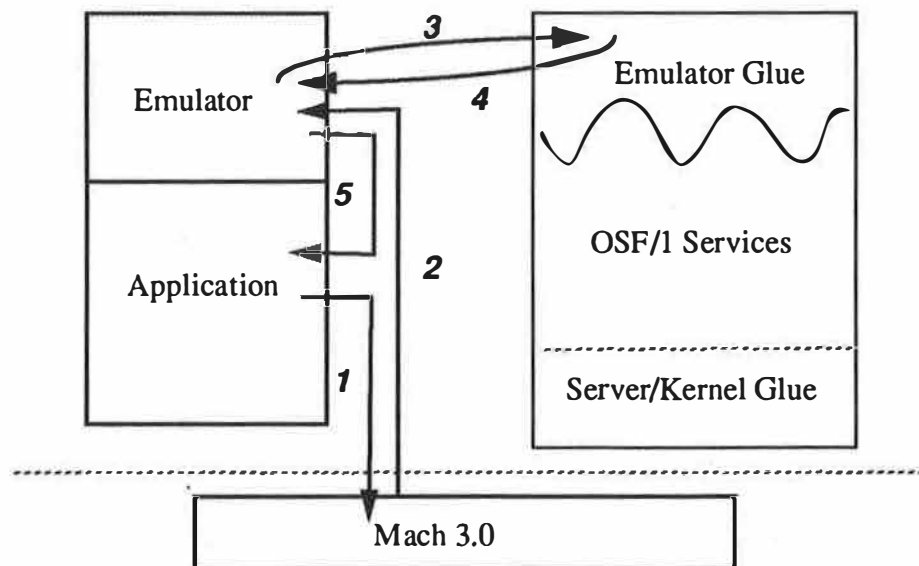
months, new mechanisms and improvements to existing mechanisms have been developed to be able to better support servers implementing Un\*x like semantics.

This paper discusses the motivation for the emulator, why it was inadequate for the needs of OSF/1, and what was used to replace it. In addition, the impact of this solution on other architectures currently being used in OSF RI is considered.

## 2. Why an emulator?

The motivation for the emulator was originally for binary compatibility. Traditional statically bound BSD binaries used hardware traps to communicate their request for a system call to the kernel. With Mach 3 however, the kernel does not know anything about BSD semantics, or even which operating system personality is running. As a consequence, these traps had to be redirected in some way. In addition, there was the apparent contradiction of style between BSD system calls and Mach's message based communication mechanisms.

The emulator is essentially part of the operating system personality. The emulator is a second program that is loaded into an unused part of the address space of every process. When the calling application does a system call trap, the emulator is entered by a simple redirection back into a pre-defined part of its address space. It can then perform the translation between a trap based interface directed at the kernel to a message based interface directed at the server. Figure 1 shows the code path of a normal system call using the emulator.



**Figure 1: Code Path of an emulated system call**

The application makes the system call in the normal way (step 1) by executing a trap instruction into the Mach kernel. The kernel looks up the trap number in a per task redirection table, saves some state on the user stack and returns back to the application (step 2) at the address in the

table. The emulator starts to process the system call and normally sends a request Mach IPC message to the server to perform the system call (step 3). Having executed the system call, the server sends a reply message to the emulator (step 4), sending back any resulting data. Finally (step 5), the emulator returns back to the address in the application that the kernel saved on the stack having stored the results of the system call in the appropriate places.

In addition to doing this simple conversion, the emulator soon became the focal point for performance optimizations. Some system calls were able to be entirely implemented in the emulator either due to the emulator caching information or due to introduction of shared data between the emulator and the server. Some other optimizations were made by having an I/O page between the server and the emulator, which eliminated copies into and out of messages.

The other advantage the emulator was seen to have was that in systems which supported shared libraries, the emulator could ultimately replace the system call stubs in libc.

## **2. Why not an emulator?**

The problems introduced by the emulator can be divided into the following categories;

- System security
- System integrity
- System resource management
- Compatibility and maintenance

It should be noted that some of these problems are OSF/1 specific, while others are Un\*x like specific and a few are generic to any personality. The root of most of the problems with the emulator is that it is untrusted due to the fact that it is completely unprotected from the application.

### **2.1 System security**

Violations of system security mean that, although the server continues to function normally, the security policy is potentially violated. This means that the operating system personality is fooled into providing services for a process which should not receive them. Using the emulator, all the Mach kernel services are necessarily exposed for direct use by the application. Because Mach services are not mediated by the server, the kernel will perform operations for anyone who presents the appropriate port. The biggest problem areas are task and thread ports which can be passed to other processes via Mach IPC. Using the current emulator based OSF/1 MK or the CMU BSD server, it is relatively easy to become root or force the server to execute system calls for privileged programs which they did not request.

One way of becoming root with the current system would be for a process to modify its emulator code/data such that some innocuous system call will do something to breach security, such as `exec()` a shell, instead of performing its original service. Then the process `exec()`s a `setuid-root` program known to use that system call. Since the emulator is inherited without its text or data being re-initialized, the `setuid-root` program will use the malicious emulator and security is broken. Although it is possible to close these types of security holes by replacing the task and reload-

ing the emulator for setuid execution, it is at the expense of performance and general increase in complexity.

Removing the emulator does not change these problems in any way but it does allow the possibility of disallowing non-privileged programs from direct access to Mach services, which is what is done. In OSF/1, Mach services are defined for use by system extensions. Consequently in initial versions of OSF/1 MK, a process needs to have the same privilege to use Mach services as it needs to load "kernel" extensions into the server. The use of the services will be re-enabled when the use of Mach services can be brought within the security policy of the operating system personality.

## 2.2 System integrity

System integrity is breached if a process can get the server to perform incorrectly and at worst, either hang or crash. There are a number of ways in which the emulator exposes the server to attack. OSF/1 MK inherited various mechanisms from the BSD server which cause problems. In addition the direct unmediated use of Mach services causes others.

There are a number of attacks that the server must defend against by using the Mach kernel primitives to manipulate resources that the server believes are under its control. In particular, the problems are processes calling `task_terminate()` instead of `exit` and processes calling `thread_create()` and then these unknown threads making system calls. Processes `vm_allocate()`ing or `vm_deallocate()`ing memory causes confusion as the server attempts to manage the processes address space to control the `brk` area and to flush mapped regions etc.

Processes can also send strange messages to the ports on which the server collects system call requests. These can either be with the correct ID and the wrong contents (for `copyin()` for example) or with additional out-of-line data that the slowly clogs the servers address/port space.

## 2.3 System resource management

Attacks of this form can come in two forms. Either the application uses more resources than it is allowed or it forces the server to use resources which could potentially cause denial of service or even to cause the server to crash or panic. Again this is not explicitly enabled by the emulator but is due to the direct use of Mach services which the presence of the emulator requires.

Exposing the external memory manager interfaces means that a malicious program could establish a mapping with an uncooperative pager which could not respond when the server accessed it. This would tie up server threads forcing the server to create more, if it wished to continue to function, until the system resources were exhausted.

The kernel itself has no mechanism for resource control and will blindly satisfying most requests to allocate resources such as threads, ports or vm address space until that resource is exhausted for the whole system.



## 2.4 Compatibility and maintenance

There are a number of problems with the emulator architecture which are perhaps more specific to OSF/1 than to other personalities as they are caused by some OSF/1 features not normally found in Un\*x systems.

OSF/1 allows for kernel loadable modules. One type of subsystem that can be loaded is a module that implements new system calls. New subsystems configure themselves dynamically when loaded, which is normally at boot time. Access to the emulator is set up when loading *init* and only system calls that are known about at this time are redirected to the emulator. This emulator vector is then inherited from task to task. Thus, the system call table is replicated for every process in the system and there is no way to update the vector for every task in the system when new system calls are added.

For almost the entire time that OSF/1 MK was being worked on in Grenoble, there were always outstanding signal bugs. Even after almost two years of development, the signal semantics of OSF/1 MK were not totally compatible with the integrated kernel, and indeed there were still some bugs remaining that would cause the server to hang. This was caused primarily by splitting the signal functionality between the server and the emulator, and then not trusting the emulator to do the job. The signals code in OSF/1 MK deviated from its integrated kernel equivalent more than any other piece of code in the system. It changed significantly in virtually every release and there were still corner case bugs after more than 12 releases. Part of the problem of delivering signals from the server is that, if the application is in the emulator, it is hard to find the real user stack to build the signal frame, as the emulator changes stacks when it is entered. When or if you do find it however, it has data the emulator has saved there and so cannot be used anyway. This is also a problem when trying to dump core.

In order to improve the performance of the emulator, a page of state is shared between the server and the emulator. This page holds various information normally found in the utask structure. This means that structure offsets are different and thus kernel extensions are not binary compatible. It also means that the server cannot trust this information and must check that it is sane before it can be used.

In addition, the kernel-to-application interface is well defined in the integrated kernel, it is `copyin()` and `copyout()`. When any functionality of a system call is implemented in the emulator, `copyin()` is implemented by the emulator knowing, in advance, what is to be copied and sending the data along with the message containing the system call request. Similarly, `copyout()` is implemented by extending the reply message that is sent back at the end of the system call with the data to be copied and then the emulator does the copy to the correct place in the user's address space. The problem with this is it means that `copyin()` only works if arranged in advance and any `copyout()`s are deferred until the end of the system call. As the size of the return message was limited, this also limited the amount of data the server could `copyout()`. This all means that any kernel extensions expecting, for example, that `copyout()` followed by `copyin()` of the same address would see the data written would not see the expected result. As a consequence, OSF/1 integrated kernel extensions are not even source compatible with OSF/1 MK.

Upgrading becomes more expensive due to the amount of code that is changed to support the emulator that has to be repeatedly applied to each new version of OSF/1. This also means that OSF/1 MK is not using OSF/1 IK code for this functionality which increases the chances of the semantics differing between the systems and introducing bugs into OSF/1 MK. The issue of kernel extensions becomes more important as the server is upgraded to OSF/1 1.2 and beyond as virtually all subsystems (NFS, System V FS, LVM, network protocols) are dynamically loaded during booting and it is unreasonable and expensive to expect people to re-code their kernel extensions for the sake of changes introduced by the emulator.

Even in itself, the emulator is a maintenance nightmare for OSF/1 MK. It is a complex piece of code involving stack switching code, port and memory management, code for signal delivery (which incidentally is duplicated in the server) and address fault recovery. There is communication between the emulator and server via shared memory, which the server cannot trust and must therefore defend against, thus forcing the complexity into the server.

Finally, there were a number of little problems. Errant programs would fail in unexpected ways if they scribbled over the emulator. Debuggers do not understand the emulator and so follow the trap redirection into a part of the address space that the programmer may not have realized (and probably does not want to know) was mapped. In OSF/1 AD, the emulator even has extra threads which are not part of the application to further confuse and add failure modes. The mapped file implementation also involved passing tokens between emulators to indicate ownership which might not be given up, so a complex time-out and revocation scheme was invented. There seemed to be excessive data copying from the servers buffers into the shared I/O page and then into the user's buffer for read(s) and write(s). The stack allocation and switching at the start of every system call seemed like another expense that was not needed. Ultimately it became too much and it was felt as though more effort was being put into maintaining the emulator than moving the system forward functionally.

### 3. What instead?

The solution was to solve two problems: The communication of a system call request to the server and correcting the implementation of copyin() and copyout() in the server.

The request from the application had to reach the server in an efficient form and as this is Mach, this should be a message. The application request for OSF/1 was in the form of a trap instruction, or equivalent, in the system call stub in libc. This meant that it had to be a kernel mechanism. The obvious choice was to send an exception to the appropriate exception port. There were numerous problems with this solution however.

The exception message Mach currently sends contains send rights to the task and thread ports of the thread that generated the exception. Sending port rights in messages is relatively slow compared to sending data alone and for system call delivery, speed is of the essence. In addition, it is certain that a system call will need to modify the state of the calling thread. With the current message format, this means the server having to call thread\_get\_state() and thread\_set\_state() for

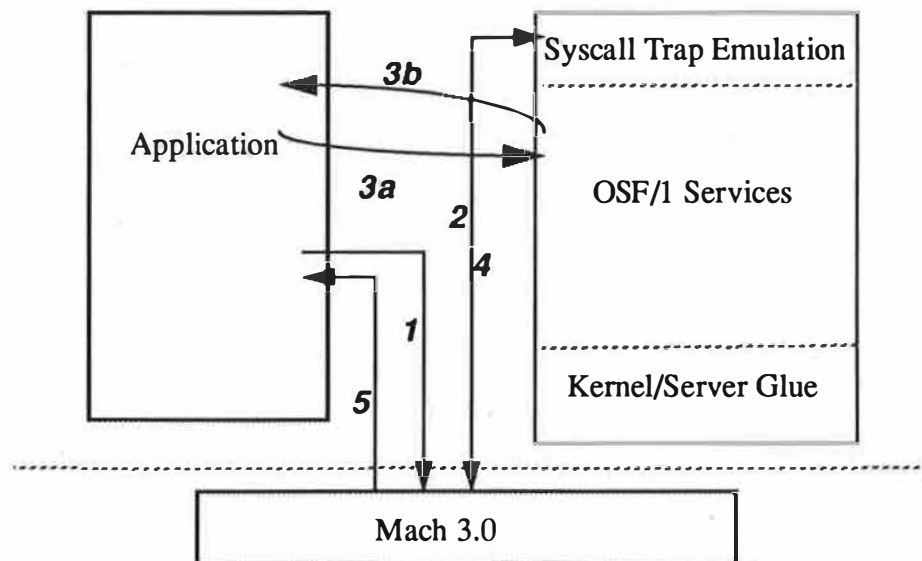
every system call. The added complication of debuggers interposing the exception port in order to intercept exception messages also means that these messages could not be sent on the regular exception port. So what with slow, port laden messages being sent to the wrong port and two additional Mach system calls being needed for every OSF/1 system call, it was clear that this would not work. However the general idea was right and this led to the exception mechanism being generalized.

In order to solve the limitations of the emulator based server implementation of `copyin()` and `copyout()`, the functions had to be made semantically identical to the integrated kernel. With the current Mach interfaces, the only available way of reading or writing to another task's address space is to use `vm_read()` or `vm_write()` respectively. Neither of these calls in their current forms are capable of doing the job for `copyin()` and `copyout()`.

The problem of `vm_write()` is simply that it only writes page sized and aligned quantities. This makes it impossible to atomically `copyout()` data that is not page aligned in both size and address in both the server and the target process. For example, to copy 40 bytes would involve the server reading the page in which the addresses it wants to modify are, modify the 40 bytes and then write the entire page back. If that page was shared by multiple threads, either in the same task or in different tasks, then there is no guarantee that one of these threads will not touch the part of the page that the server does not wish to change and if it does, the server's page write would overwrite these modifications.

The problem for `vm_read()` is simply that it returns the data out-of-line. (It also page aligned in both size and address, but that is not a semantic problem). As a consequence of this, the data is never where it needs to be and so it must be copied. The original data returned out-of-line must then `vm_deallocate()`'d. `copyin()` is not a function that can tolerate making two Mach system calls as it is called too frequently.

The result of making these two changes, adding the new exception mechanism and fixing `copyin()/copyout()`, is that the architecture looks remarkably like that of the integrated kernel, with exception that the OSF/1 services are in user space rather than being bound in with the Mach kernel. The interface between the application and operating system personality, OSF/1 in our case, is back to being trap from the application to the personality and `copyin()/copyout()` from the personality to the application.



**Figure 3: Code path for a system call without an emulator**

The application makes the system call request in the normal way (step 1) by executing a trap instruction into the kernel. The kernel generates an exception message which it sends to a port that the server has set up especially for handling system call requests (step 2). The server then processes the system call in the same way as in the integrated kernel, calling `copyin()` (step 3a) or `copyout()` (step 3b) whenever it needs to get or return data to the application. Finally the server returns any modified register state to the kernel (step 4) in the reply to the exception message, and the kernel resumes the thread having restored it (step 5).

#### 4. A generalized exception mechanism

Having initially tried having the kernel send specialized messages when traps occurred, it was found that the code used in the kernel was remarkably similar to that to send exceptions. The motivation for considering a separate interface was simply performance, i.e. to avoid sending ports. From the implementation however, it was clear that what was trying to be achieved was to simply raise an exception on the occurrence of a trap instruction.

The current Mach exception mechanism treats all exceptions alike, in that the kernel generates a message and sends it to a thread's exception port, if it has one, or the task's exception port, if not. The message that is sent is the same in all cases and it is intended simply to identify the task and thread that caused the exception. The new mechanism changes this in two fundamental ways.

Firstly, to obviate the need for a server to use `thread_get_state()` and `thread_set_state()` when exceptions occur, new versions of the exception message were defined in which the state is sent and the modified state returned with the reply, which the kernel will set before resuming the thread. In addition, one of the new interfaces does not send the task and thread port of the thread causing the exception, making the assumption that the thread can be determined by the port that

the incoming message appears on. The MiG `exc_server()` routine calls either `catch_exception_raise()`, `catch_exception_raise_state()` or `catch_exception_raise_state_identity()`, depending on the ID of the exception message the sent by the kernel. The new exception message interfaces are listed in Appendix A.

Clearly, the state needed for dealing with system call exception messages would not necessarily be the same state as that for dealing with arithmetic or address faults. Consequently, a way of associating what flavor state is required to be sent in response to a given exception type was introduced.

Secondly, to avoid confusion for debuggers that wish to interpose exception ports for the applications that they are controlling, you would not want to send system call exceptions, which the debugger is not interested in and probably couldn't handle, to the same port as arithmetic faults, which the debugger is interested in and can handle. Due to this, and since it is likely that some servers will want to use different threads to handle the different types of exception, a new interface makes it possible to associate different ports to different exception types for a given thread/task. The behavior specified determines the type of message the kernel will send and the flavor indicates the flavor of thread state the kernel sends with that message. Obviously, if the behavior is `EXC_DEFAULT` indicating that `catch_exception_raise()` is to be used, then the flavor is ignored. The new exception port manipulation interfaces are listed in Appendix B.

The types of exception supported are those returned currently by `catch_exception_raise()` in the exception parameter, plus the additional types `EXC_TYPE_SYSCALL` and `EXC_MACH_SYSCALL`. However it is expected that certain architectures will wish to define more.

In addition to the current thread state flavors, a new flavor was added, `THREAD_STATE_SYSCALL`. The definition the contents of this state is machine dependent but the intention is that it contains a small subset of registers adequate for handling most system calls for most personalities. For the i386, this register set is currently 5 registers (as opposed to 17 for the full set).

Note that

```
task_set_special_port(task, THREAD_EXCEPTION_PORT, new_port);
```

is equivalent to

```
task_set_exception_ports(task, EXC_MASK_ALL & ~EXC_MASK_SYSCALL,  
                        new_port, EXCEPTION_DEFAULT, 0);
```

The same will apply to the other task/thread get/set exception calls. However a backwards compatibility problem has been introduced as the format of the `catch_exception_raise()` interface has changed (for a completely unrelated reason). So, in our kernels, setting the exception ports with `task_set_special_port()` will generate the old `catch_exception_raise()` format message but doing the equivalent operation with `task_set_exception_ports()` described above, the new `catch_exception_raise()` format is used, thus it was possible to achieve both source and binary compatibility.

Delivery of exceptions in the kernel uses exactly the same algorithm. When an exception occurs, the kernel tries to deliver the exception message to the thread exception port for that exception type. If that does not exist or returns a value other than KERN\_SUCCESS then the task exception port for that exception type is tried. If the exception is not handled after both possibilities have been tried, the kernel terminates the task, as before. Note it is perfectly possible and reasonable that the task and thread ports for a given exception type have different behaviors and flavors set.

## 5. Solving the copyin()/copyout() problem

Two solutions to this problem were explored. One was relatively conventional in Mach terms and the other rather controversial. The first was to 'fix' vm\_read() and vm\_write() such that they could be used in such performance critical code as copyin() and copyout(). The second involved the server mapping arbitrary addresses in the target process.

### 5.1 Using vm\_read() and vm\_write()

The change to vm\_write() was simply to allow the specification of addresses in both the source and the target tasks that are not page aligned and to allow quantities of data that are not multiples of a page in size. In fact a large number of people had seen this as a bug in the specification of vm\_write() (there was even a bug report against OSF/1 1.1 for this) so this seemed like an uncontroversial change. It had the added advantage that it did not effect existing callers of vm\_write(). Clearly if unaligned data was specified, there was no possibility for the kernel to perform any VM manipulations which would avoid physical copying of data; however, the implementation was careful to ensure that if correctly aligned data was specified then the appropriate VM magic would occur.

The change for vm\_read() was slightly more awkward. The problem was caused by the fact that vm\_read() returns the data out-of-line. For copyin() semantics this has two ramifications; the first being that the data is not where you want it thus causing a copy, and the second being that the returned data needs to be vm\_deallocate()'d after use thus incurring the extra cost of another Mach service. The solution was to introduce a new class of Mach interface; one which returned out-of-line data in a buffer that the receiver already had mapped.

This was done by introducing a new interface, mach\_msg\_overwrite(). The semantics are exactly the same as mach\_msg() with out-of-line data with the exception of final delivery to the receiver. The receiver specifies an already mapped address and range in which the kernel is to put any out-of-line data that the message may contain. If the specified buffer is too small, then the call to mach\_msg\_overwrite() fails. Once this call existed, it was simply a case of changing MiG to use it. The resulting interface was vm\_read\_overwrite() which is semantically identical to vm\_read() with the exception that the call provides the buffer. Using this interface, there was no need for the bcopy() (as the kernel did it or the equivalent) but more importantly, the vm\_deallocate() was removed from the copyin() path.

## 6. Using `vm_remap()`

There was still concern that one Mach system call per `copyin()` or `copyout()` was still one too many, especially considering that performance of the system would be compared against the integrated kernel, whose implementation of these functions normally amounted to the equivalent of a `bcopy()`. To solve this problem, a new interface was added to Mach which seemed to break one of the rules of Mach interfaces, that of being location transparent. The interface, called `vm_remap()`, maps arbitrary regions of one task's address space into the caller's address space.

The main issue is that the interface is not location transparent in that if the target task was not on the same host as the caller, then the call would fail. The reason that this will fail is not an inherent problem with the interface as it is specified, but it is more due to the fact that most pagers do not know how to manage multiple kernels (the netmemoryserver being the obvious exception). There is the same problem with `default_pager_object_create` being used by two clients of `vm_map` on different hosts. Also, there is no mechanism to allow the kernel managing the source task to inform the kernel that is managing the target task of the memory objects that represent the address range in the source task. This problems could probably be fixed but it would involve some inter-kernel communication. `Vm_remap()` will work on multiple kernels within a NORMA domain even across different nodes as the NORMA XMM performs basically the same job as the netmemoryserver within the NORMA domain and the NORMA kernels cooperate with task address space management, as this is needed to support `norma_task_create()`.

The call to `vm_remap()`, listed in Appendix C, is similar to `vm_map()` except that instead of a `memory_object` port being supplied to the kernel, a task port and a virtual address range is specified. The kernel then maps the underlying objects of the source address range into the target task's address space. The implementation of `vm_remap()` simply replicates the kernel data structures in exactly the same way as would have happened if one of the tasks was specified as the parent task to `task_create()` and this portion of memory was marked as `VM_INHERIT_SHARE`. The main reason for this choice was to avoid any new kernel data structures or new vm relationships which would increase the complexity of the VM generally.

The disadvantage of mapping the underlying objects instead of somehow sharing address maps, is that deallocating the addresses in the application does not invalidate the mapping in the server. Consequently the server has to be very careful when it deallocates memory on behalf of the application to ensure that the equivalent server mapping is deallocated also. Even worse is that if an application used Mach primitives directly to deallocate this address range, the server would never get to know about the change and any subsequent system call specifying this address range would use the now stale mapping in the server. However to really share address ranges would involve a considerable change to the kernel management of address maps which was not consider either desirable or necessary.

Two different caching schemes were implemented in the server to take advantage of existing mappings. One was a system wide cache and the other was a task local cache. Both were prototypes and need further work in one area or another.

The system cache consisted of a hash table of mappings which were indexed by application

virtual address and PID. A LRU (Least Recently Used) list of mappings is maintained for garbage collection when the hash table gets full. This had the advantage that on a quiet system, a single application could use as much of the cache as it liked but on a busy system, an application that used buffers distributed throughout its address space would use an excessive number of cache entries and thus cause better behaved applications to garbage collect the cache to find a free slot.

The second caching scheme was to maintain a simple linked list of mappings per task. Limits on the number of mappings and the amount of server virtual memory were applied to the task to prevent a task from using up server resources excessively. Two hints were maintained into the linked list based on address to speed the lookup.

Both caching schemes seemed to perform similarly with cache hit rates in excess of 90%. The system wide cache was marginally slower but this was caused by two deficiencies, one in each cache implementation. The system wide cache did not implement hints (which in the per task scheme had a very high hit rate, thus obviating the cache lookup completely) and the per task cache was not thread safe so avoiding lock overhead. I doubt if there would be a significant difference in performance if these two problems were fixed.

## 7. The resulting system

As a result of using per thread exception ports, a large amount of restructuring of internal data structures occurred in the final implementation. In fact, to simplify other things, the original OSF/1 kernel internal data structures were reintroduced, that is the Mach 2.5 task and thread structures. The system was very stable remarkably quickly, passing VSX and VSE to the same level as OSF/1 MK V4.1 within a few weeks of getting to multi-user mode and ran AIMIII to 250 users. This is compared to the emulator based server which took much of a year to get to the same level of robustness and functional completeness. The main reason for this was the significant increase of reuse of mature OSF/1 Integrated Kernel code rather than writing new code to implement the same functionality, as was done for the emulator. The code reuse was so great that even the integrated kernel's function `syscall()`, which handles system call setup, could be virtually entirely reused, with 15 extra lines of preamble, 2 postamble and a name change to `catch_exception_raise_state()`.

As the internal environment became indistinguishable from the integrated OSF/1 system, virtually all the subsystems, e.g. ufs, vfs, net, netinet, etc., were reused totally unchanged. This in turn has made the upgrade of the 1.0.4 based version of the server to be based on OSF/1 1.2 to be achievable in a far shorter time.

The table below shows performance ratios of various versions of the no-emulator server compared to OSF/1 MK V4.1, which is the latest emulator based version of OSF/1 MK. The first 6 benchmarks are micro-benchmarks consisting of a 10 iterations of 1000 loop tests of either one, or a small number of system calls. The choice of system calls was based on the number of `copyin()/copyout()` operations they performed rather than frequency of use and thus are not representative of the overall performance that could be expected of the system, but more indicate the behavior of the mechanisms that are the subject of this paper. The AIMIII tests at 24 and 32 users give a better



idea of overall system behavior. All the tests were performed using the same kernel which was an

	OSF/1 MK 4.1	vm_remap (system cache)	vm_remap (task cache)	vm_read_over write
getuid	1.00	1.04	1.04	1.05
read/lseek	1.00	0.73	0.71	1.06
File copy	1.00	0.81	0.88	0.88
uname	1.00	1.00	0.94	2.29
exec	1.00	1.06	1.04	1.37
sigaction	1.00	12.99	11.95	23.74
AIMIII 24 Users	1.00	1.04	1.01	1.42
AIMIII 32 Users	1.00	1.03	1.00	1.48

Performance Ratios of OSF/1 MK V4.1 using the emulator Vs. No-emulator servers

extended NMK13.3 with support for the no-emulator servers.

From the performance perspective, the new architecture was about performance neutral with OSF/1 MK V4.1, the latest version which used the emulator. This is encouraging as V4.1 is a product of many performance optimizations implemented over a year whereas there has been no effort put into optimizing any part of the no-emulator server yet. As there are known performance problems in the no-emulator server, this augurs well for the final performance of this server.

The `vm_read_overwrite()/vm_write()` still seems to have some performance problems. Admittedly there are some known kernel optimizations that can be made to these calls, but it is still clear that the server can afford the Mach call for every `copyin()` and `copyout()` operation. The high cache hit rate seems to be helping considerably for the `vm_remap()` versions of the server but neither caching strategy seems to be a clear winner judged simply on performance. Also, as neither caching scheme was fully implemented, it seems premature to make a judgement between them until they have been.

## 8. Suitability for other architectures

The main architectures that OSF has to consider, beyond traditional uniprocessors and symmetric shared memory multiprocessors, are multicomputers, otherwise known as NORMA (NO Remote Memory Access) machines, and Clusters of cooperating machines sharing a single system image. Both systems have a set of similar problems when it comes to distributing services, even though the details of the solutions may differ slightly. It was important that any change of server/kernel architecture was usable by systems such as these.

One of the main differences between this architecture and that using the emulator is that, as with the integrated kernel, the applications system call request enters the TCB (Trusted Computing Base) immediately (the emulator is not a trusted piece of code). As the only way of getting to the system call service point is via a single Mach port, the system call exception port, this implies that all system calls are, at least initially, handled by the same server. At first glance, this may not look as if it matches well with the idea of distributed operating system services, however having a protected local entity has many benefits that the emulator solutions could not properly provide.

On each node of these systems there can be a local agent, which acts as the trusted router of

system call requests for all the tasks on this node. This allows the agent to play its part in distributed database management (for things like credentials, process management, etc.) like the untrusted emulator never could. In addition, the local agent can also play the part of a trusted cache manager by performing read ahead on file data not only to bring the data off the disk, but also to bring it local to the task so that a subsequent read would not have to leave the node to be satisfied. The agent will also at least have to participate in, if not initiate, process migration between nodes. None of this could be satisfied if the agent was not a trusted piece of code.

This does not mean however that an application may never communicate directly with a remote server. Once the problems of exposing Mach interfaces to applications (briefly described in Section 2) have been solved, a new libc could be implemented which used the services of both a local agent and remote servers, using the agent as the distributor of access means (i.e. ports) to the remote services.

Both the next version of OSF/1 AD (for multicomputers) and the version being designed for clusters are being based on the no-emulator technology.

## 9. Conclusion

The work is still not fully implemented in a final system. The 1.0.4 prototype was missing a number of key optimizations and small pieces of functionality were not implemented as they were seen as not to be relevant to this proof of concept. However, it is being used as the basis for the OSF/1 1.2 based version of the server, which will be the first version of OSF/1 MK that can be seriously considered for productization and is the first version that is expected to be able to attain C2 and B1 security levels. This server is also being used as the base for all OSF RI operating systems technology, including for multicomputers and clusters.

There is still more work to do examining the `vm_remap()` vs. `vm_read_overwrite()` versions of the server to determine if the difference in performance between the two systems is inherent in the design or could be solved with a better implementation. The new servers are very robust considering how long they have been running which is undoubtedly due to the increased use of mature OSF/1 integrated kernel code. In addition, the system's semantics are now as close to those of the integrated kernel that they can be, given that the system is now in two parts, the kernel and the server.

The performance data is promising but also implies that there will probably not be any performance revolutions using this architecture, however it is believed that we can exceed the performance of the emulator based servers once certain optimizations have been implemented.

## 10. Acknowledgments

I would like to thank all the members of the OSF/1 MK team in the OSF Research Institute, Grenoble for their contributions to this work including their ideas and the efforts implementing the final design. In particular I would like to thank Michael Condict and Philippe Bernadat for many passionate design discussions, François Barbou des Places, Éamonn McManus, David George, Christian Bruel and Rod MacDonald for their vital help in implementing the system. I would also like to thank all the OSF Research engineers whose scepticism kept us honest.

## Appendix A

### Exception Message Interfaces.

```
kern_return_t
catch_exception_raise(
    mach_port_t      exception_port;
    mach_port_t      thread;
    mach_port_t      task;
    exception_type_t  exception_type;
    exception_data_t  code;
    mach_msg_type_number_t  code_count);

kern_return_t
catch_exception_raise_state(
    mach_port_t      exception_port;
    exception_type_t  exception_type;
    exception_data_t  code;
    mach_msg_type_number_t  code_count;
    thread_state_flavor_t  *flavor;
    thread_state_t      *in_state;
    mach_msg_type_number_t  in_state_count;
    thread_state_t      *out_state;
    mach_msg_type_number_t  *out_state_count);

kern_return_t
catch_exception_raise_state_identity(
    mach_port_t      exception_port;
    mach_port_t      thread;
    mach_port_t      task;
    exception_type_t  exception_type;
    exception_data_t  code;
    mach_msg_type_number_t  code_count;
    thread_state_flavor_t  *flavor;
    thread_state_t      *in_state;
    mach_msg_type_number_t  in_state_count;
    thread_state_t      *out_state;
    mach_msg_type_number_t  *out_state_count);
```

## Appendix B

### Exception Port Manipulation Interfaces.

```
kern_return_t
thread_set_exception_ports(
    mach_port_t          thread;
    exception_mask_t      exception_types;
    mach_port_t          exception_port;
    exception_behavior_t  behavior;
    thread_state_flavor_t flavor);

kern_return_t
thread_get_exception_ports(
    mach_port_t          thread;
    exception_mask_t      exception_types;
    exception_mask_array_t old_exception_types;
    mach_msg_type_number_t *old_exception_count;
    mach_port_array_t    old_exception_ports;
    exception_behaviorarray_t behavior;
    thread_state_flavor_array_t flavor);

kern_return_t
thread_swap_exception_ports(
    mach_port_t          thread;
    exception_mask_t      exception_types;
    mach_port_t          exception_port;
    exception_behavior_t  behavior;
    thread_state_flavor_t flavor;
    exception_mask_array_t old_exception_types;
    mach_msg_type_number_t *old_exception_count;
    mach_port_array_t    old_exception_ports;
    exception_behaviorarray_t behavior;
    thread_state_flavor_array_t flavor);
```

## Appendix C

### Arbitrary Memory Mapping Interface.

```
kern_return_t
vm_remap(
    mach_port_t
    vm_address_t
    vm_size_t
    vm_address_t
    boolean_t
    mach_port_t
    vm_address_t
    boolean_t
    vm_prot_t
    vm_prot_t
    vm_inherit_t

    target_task,
    *target_addr,
    size,
    mask,
    anywhere,
    source_task,
    source_addr,
    copy,
    prot,
    *max_protection,
    inheritance);
```



# A Fast and General Implementation of Mach IPC in a Network

H. Orman      E. Menze III      S. O'Malley      L. Peterson \*

Department of Computer Science  
University of Arizona  
Tucson, AZ 85721

## Abstract

This paper describes an implementation of the Mach IPC abstraction on a network. Our implementation, called Mach NetIPC, is done in the context of the *x*-kernel, which provides a networking subsystem for Mach. The paper motivates the design choices we made, describes the *x*-kernel protocol graph that implements the design, and reports on the performance of the resulting system.

## 1 Introduction

Mach IPC is the communication abstraction of the Mach operating system. It supports a rich semantics: multi-part, typed messages are delivered reliably and in sequence to ports; tasks hold, and can transfer to each other, the right to send to, and receive from, ports; and tasks are notified when the status of ports they hold a right to changes (e.g., the holder of a send right is notified when no task currently holds a receive right). Although straightforward to implement in the microkernel of a single machine, extending Mach IPC semantics to a network environment—we call such a system Mach *NetIPC*—is problematic. Multiple host architectures, individual node failures, lost messages, network partitions, and variable network delays all combine to make the implementation of NetIPC a difficult proposition.

This paper describes an implementation of Mach NetIPC in the context of the *x*-kernel—a protocol implementation environment that has been incorporated into Mach [4, 7]. Our implementation of Mach NetIPC is not the first; there have been several earlier implementations of the Netmsgserver [5], and an in-kernel implementation tailored for NORMA architectures [1]. Because of our use of the *x*-kernel, however, our implementation has the following distinct advantages.

- The *x*-kernel supports the decomposition of complex protocols into “microprotocols” that are small, easy to understand, implement, debug, and tune. This has helped us to manage the inherent complexity of Mach NetIPC.
- The *x*-kernel supports configurable protocol graphs which make it possible to optimize the set of protocols used for different architectures, including tightly-coupled distributed memory multiprocessors, workstation clusters, and arbitrary collections of hosts connected by the Internet.
- The *x*-kernel protocol graph can be partitioned across multiple protection domains. This allows partitioning of the Mach NetIPC protocols between the microkernel and a user task, depending on how one wants to trade performance against trust. In other words, when building a secure version of Mach where microkernel minimization is important, one can place all the network protocols in a user task; when building a high performance version of Mach (perhaps for a multiprocessor) one can locate the Mach NetIPC protocols in the kernel.

---

\*The work presented here was supported under DARPA grant DABT63-91-C-0001.

This paper is organized as follows. Section 2 motivates our design by examining various issues related to a network environment. Section 3 then describes the *x*-kernel protocol graph that implements Mach NetIPC and Section 4 reports on the performance of our implementation. Finally, Section 5 discusses the relationship between our implementation and proposed changes to Mach IPC, and Section 6 presents some concluding remarks.

## 2 Design Issues

This section identifies the difficulties in extending Mach IPC over a network, and reports the design choices we made to address them. The discussion motivates the protocols described in the next section.

### 2.1 Decentralization

An obvious problem in extending Mach IPC to run over a network is that information about ports—in particular, what tasks hold send and receive rights—must be maintained in a distributed fashion. In this context, the key question is what information does any single node need to maintain, and does this limit how large the system can grow.

A goal of our design is that it scale arbitrarily. That is, rather than assume that the set of tasks communicating with each other are restricted to a limited set of nodes that are known *a priori*, or restricted to a single network (e.g., a LAN), we assume that Mach NetIPC can be used across an arbitrary collection of nodes. Stated more precisely, if we consider nodes A and B to be *linked* when a task on node A holds a send right a port and a task on node B holds a receive right to the same port, then we expect the transitive closure of all linked nodes to be arbitrarily large—tens or hundreds of thousands—and possibly spread over the Internet.

Note that this definition of scalability is orthogonal to how Mach NetIPC scales across two other dimensions: (1) the number of tasks running on different nodes that all hold rights to any one port, and (2) the number of ports that tasks on any one node will hold rights to. In both of these cases, we would consider anything over a few hundred to be unusually large. This seems to be a fundamental limitation of Mach IPC, and is not specific to its implementation in a network.

The main implication of this goal is that we cannot expect to maintain a single, complete database of port rights, even if that database is physically replicated throughout the network. Instead, we can only expect each node to maintain information about those ports to which its tasks hold rights. In particular, each node that holds a send right knows the node that currently holds the receive right for the port (but not about the other senders), and the node that holds the receive right keeps track of all the nodes that currently hold a send right. A second implication is that our implementation cannot depend on the existence of a network broadcast facility.

### 2.2 Node Failure

It is important for Mach NetIPC to survive in the face of simple node failures, meaning that nodes that were communicating with the failed machine must eventually remove all state associated with the communication, including all port rights. We define a simple failure as one that exhibits crash semantics, that is, a node fails silently without transmitting any extraneous messages [10]. To meet this requirement, we have designed a failure handling system that is consistent with the fully decentralized communications model presented in the previous section.

We have adopted a “retry until reboot” strategy that dictates the equivalence of node reboot with node failure notification. When a node fails, all outstanding connections with that node will continue to retry until they detect a reboot. A confirmed reboot will cause deletion of all port rights associated with the node. The system assigns a unique identifier to each incarnation of a node, and that identifier is communicated to interested parties. The incarnation number of a remote node is used to provide an index of liveness for local database items and communication channels.

An additional mechanism, an administrative shutdown, is needed to handle the problem of node failure without reboot (e.g., removal of the machine to a warehouse). Using this mechanism, a



systems administrator could inform Mach NetIPC of a node failure, causing the software would take care of all required notifications.

Our approach to node failure avoids the problems associated with incorrect failure notification and network partitioning. This is in contrast to systems that depend on timeouts (or heartbeat messages) to detect failure. In such systems a slow machine or one separated by a network partition may be mistakenly declared failed, which implies that the node must be killed (at a minimum, all of its network connections are severed) in order to ensure consistency. In normal operation using our approach, there is no possibility of a live node being declared dead, and thus, no need for the NetIPC code to kill a live node to insure the consistency of the port rights database. (Of course an incorrect administrative shutdown request could lead to a false notification, but we consider fixing this to be the responsibility of the issuer of the shutdown.) Our approach also inflicts no performance penalty in the common case where no nodes fail. Finally, the correctness of this approach is almost solely dependent upon the correctness of one simple protocol – an easily analyzed piece of code.

This failure model is different from that found in the Mach 3 NORMA IPC implementation[1]. The NORMA code assumes that there is a fixed and known collection of communicating nodes, that network partitions do not happen, and that there is no independent node failure (e.g. if one node in the group fails all nodes fail). While this approach has performance advantages, it does not scale well beyond distributed memory multiprocessors and tightly coupled workstation clusters. The Netmsgserver [5] uses reboot notifications and liveness checks to deallocate port rights, but the method is susceptible to inconsistent views arising from network partitions.

A disadvantage of our approach is a lack of timely notifications. We trade quick detection of probable failure via timeouts for certain notification of reboot via incarnation numbers. This leaves the options for fast fault detection and recovery to the realm of higher-level protocols. We believe that the administrative shutdown mechanism provided is sufficiently powerful to support such improved fault detection without requiring any change to the underlying Mach NetIPC implementation. In any case, we believe that Mach NetIPC is the wrong level to implement sophisticated failure detection mechanisms and is certainly the wrong level to act upon notifications generated by such mechanisms.

## 2.3 RPC: The Key to Efficiency

Discussions with Mach IPC designers indicated the importance of the Remote Procedure Call (RPC) concept in overall Mach client-server efficiency. Therefore, one of our first design decisions was to put an RPC protocol at the heart of Mach NetIPC. Previous *x*-kernel work had led to an efficient RPC protocol based on Sprite RPC [9, 3, 2]. This was an obvious starting point for our design. The more complex problem, however, was how to match this to Mach IPC; the communication abstraction does not have a direct match to RPC semantics, and there is some latitude possible in choosing a mapping.

Mach messages that use a send-once right for the reply port fit well with RPC, partly because there is no persistent state for the reply port. The receiver of the message can reply using the send-once right, and if this is one within one processing context, no database entries need be made. Similar consideration led to limiting the RPC definition to messages that do not carry additional port rights. This makes it possible to define a close match between the *x*-kernel implementation of the protocol and the microkernel support for RPC messages. A single thread is used on the initiating side for the request/reply, and a single thread is used on the responding side.

Non-RPC messages are handled efficiently as uni-directional messages; an empty reply message unblocks the sending thread. An eventual reply using the Mach reply port is handled as a separate, unrelated RPC (with a null reply) going in the opposite direction. This unification of the non-RPC case with the RPC case simplified construction of the protocol graph and the Mach NetIPC program.

## 2.4 Port Right Transfers

The design goals for port right transfers were fourfold: a correct algorithm, minimum overhead for the most common transfers, a deterministic algorithm, and the avoidance of any centralized database that would complicate overall robustness.

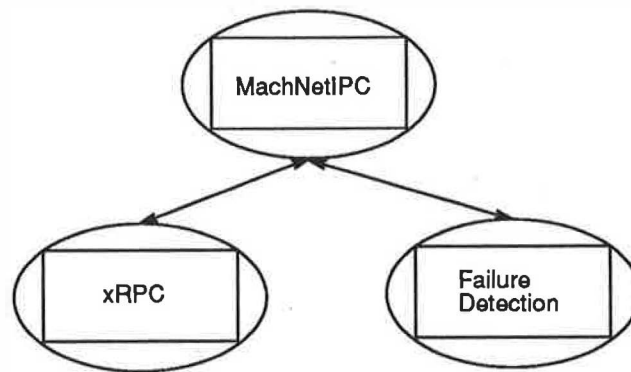


Figure 1: High-level view of protocol dependencies

A Mach port has one receive right and an unlimited number of send and send-once rights associated with it. These rights can be distributed arbitrarily across the network graph, and the method of distribution is to embed port rights in Mach messages sent between nodes. In most transfers, only the receiving node and the sending node participate in transferring the port right, and it is handled as a normal side-effect of message transfer. One exception occurs when an embedded send right has a receiver node that is neither the message sender nor the message receiver. Another occurs when an embedded receive right is associated with one or more sending nodes that are outside the message send/receiver pair. Each case invokes a multi-party protocol to keep all senders up-to-date on the current receiver for a port, and to ensure that the receiver knows all sending nodes for a port.

Moving a receive right involves coordination of the old and new receiver and notification of all senders of the new receiver address. Moving a send right involves only three parties: the old and new sender and the receiver.

### 3 Description of the Protocols

The design details of the protocols used in our Mach NetIPC implementation are presented in this section. This implementation consists of 15 protocols divided into three major groupings: Mach specific protocols (Mach NetIPC), failure detection protocols, and the *x*-kernel RPC protocols (xRPC). A coarse view of these groupings and their relationships is given in Figure 1.

#### 3.1 Mach NetIPC

The Mach NetIPC implementation consists of four modules, each of which behaves as a protocol: the Mach message handler (MNIPC), the port manager (PORTM), and the port transfer protocols (RRX and SRX). These modules depend on the xRPC protocol group and failure detection mechanism for transport services and reboot notification, respectively. This section discusses the design of the first modules and how their control structure contributes to the overall efficiency of the mechanism.

The main function of this protocol group is to represent Mach messages in a form that carries enough information to be usable on a remote machine. The basic Mach message is augmented with a header that describes the sending machine architecture, the type of the message (RPC request, RPC reply, RPC-request-no-reply, or port management), a message sequence number (unique to the sending node), and the network representation of the Mach local and remote ports. The network port representation is globally unique, based on the IP address of the node that creates it. Simple Mach messages have this header prepended, and no other modification is made by the sender.

Complex messages—those containing embedded ports or out-of-line data—have a list of network port descriptors and a byte array consisting of the concatenation of all out-of-line data added to the

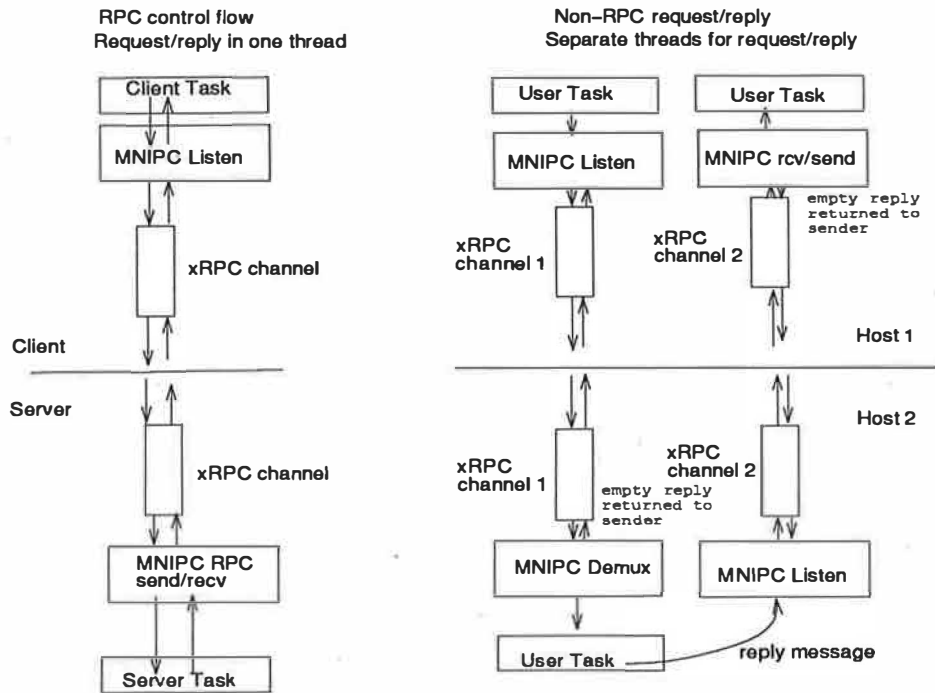


Figure 2: Matching Mach messages to *xRPC* sessions

Mach message. In addition, embedded port rights are replaced by their index in the network port list.

A major design goal was to get a close match between a Mach IPC port used for a Mach RPC (send/receive options) and the use of an *xRPC* channel. In fact, all actively used Mach ports that utilize our network services have a mapping to such a channel. The challenge was to remove as much overhead as possible from the main path, thus tying the performance of Mach NetIPC as closely as possible to that of *xRPC*. To do this, we needed to carefully define which Mach messages were acceptable as RPC's. The definition we have used is that a simple message with a send-once right as the reply port is an RPC that will receive "fast-path" treatment. These messages match the RPC semantics because the server side is stateless. That is, because the send-once right is consumed by the reply, the server side retains no state information that would have to be coordinated with the client.

When the Mach NetIPC module recognizes a message as an RPC request, it attaches a header marking it as such and sends it to the *xRPC* protocols using *xCall*, a blocking send/receive interface of the *x*-kernel. For this to preserve the sequencing semantics of Mach IPC, and for the reasons stated above, we require that the reply port of the Mach message specify a send-once right. When the *xCall* completes, the reply message is translated from network form to local form, and then delivered to the send-once right. This is illustrated in the lefthand side of Figure 2. In this diagram, a single thread is used to receive the Mach message and process it down through the protocol graph to the network; the same thread returns the reply. Similarly, on the server machine, a single thread receives the message from the network, delivers it to the server task, and returns the reply.

The send-once reply ports have a lightweight network representation that is simply a 64-bit identifier generated as a monotonic counter on the originating node. In theory, no identifier is needed because the reply is associated with the *xCall* reply message parameter, but not all RPC requests result in usable RPC replies, as explained below. No other database operations are made for send-once rights in RPC's in the normal case.

On the server side of the RPC, the request message is received for dispatching from *xRPC* by the

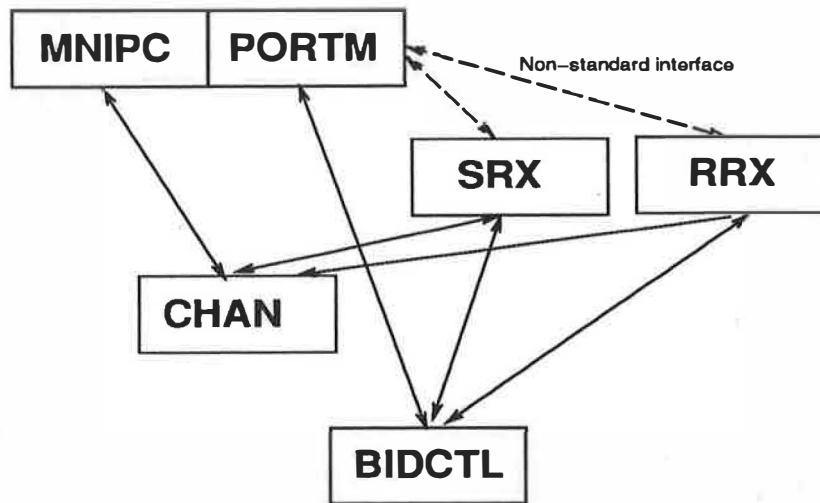


Figure 3: Mach NetIPC immediate protocol neighbors

*xCallDemux* *x*-kernel interface routine. A Mach port for the reply is assigned from a cache of pre-allocated ports. The request is translated from network form to local form, and it is delivered using the *mach\_msg* options for RPC (MACH\_MSG\_RCV and MACH\_MSG\_SEND). This can complete either by a timeout or by a successful message receive. In the most efficient case, a non-complex Mach message is received with a reply port of MACH\_PORT\_NULL. It is translated to network form and sent to the network. This work is accomplished without any thread switches.

In the less efficient case, either a timeout occurs or the message is complex. In this case, an empty xRPC reply message is returned. The reply port is entered into the network port database, and, if a timeout has occurred, a thread is started to listen for the Mach reply. The originating side, upon receipt of the empty message, similarly enters the reply port into the network database. When the reply message arrives, it is matched against the database to locate the Mach reply port, and a *mach\_msg* operation completes the transaction. This is illustrated in the righthand side of Figure 2. In this case, two threads are used on each node: one for the request, and one for the reply. In addition, separate channels in the xRPC protocol stack are used; an empty reply message completes the transaction in each direction.

When Mach NetIPC receives a Mach message that is not “RPC-style,” it assigns an xRPC channel for the remote Mach port, attaches a non-RPC marker to the header, sends the message, and does not look at the reply, because it should be empty. The empty reply serves as acknowledgement of message receipt, allowing any locked resources to be released. This is also illustrated by each half of the righthand side of Figure 2.

Our initial implementation did not use the lightweight representation of send-once rights, and the request and reply were treated as separate transactions. In this case, a port database entry was made for each send-once right, and a separate thread handled the reply. The result was that tests using send-once rights used 25% more time than tests using send rights. This necessitated the implementation of a lazier approach to send-once rights, which greatly improved performance. However, it complicated what was an otherwise straightforward implementation.

Mach NetIPC has two modules that are invoked for every message: one that handles the semantics of the data portion of Mach messages and one that maintains the port database. The latter module is an impostor *x*-kernel protocol: during initialization it replicates the Mach NetIPC object structure and changes its name. Thereafter, only messages or transactions involving the network representation of Mach port rights will go through the second module, known as PORTM, the port maintenance module. The first Mach NetIPC module, the data module, communicates with *portm* via a set of subroutine calls for creating, modifying, and deleting network port rights. When a port manipulation involves more than two machines, PORTM will use the SRX or RRX protocol to transfer the send or receive right, respectively. These four protocols and their relationships are illustrated in Figure 3.

The port maintenance module supports port locking (to limit port mobility while messages are in transit), no-more-senders, and port death. Port death occurs either when a port is destroyed or deallocated on a functioning node, or when a node holding a receive right fails.

To support no-more-senders notifications, a receiving node keeps a per port list of the addresses of all sending nodes. In this list, a “network make-send” count is maintained for each sender. This count reflects the number of send rights to the port that have been transferred to that particular node. When a sending node reports a no-more-senders condition for a port, it includes its “network make-send” count. The receiving node can compare the count in the message with the count in its database to discard stale messages with lower counts. The synchronization in the port transfer protocols can result in such stale no-more-senders messages, even with an ordered network. When all sending nodes have reported no-more-senders, and the send-once count is at zero, the receiver can destroy the port. This is, of course, a re-enactment of the accounting done by the local microkernel.

This organization suggests that the port maintenance module could be a separate protocol, and this is true but for one caveat. The port maintenance module performs operations on Mach port right names, and these must be the same names used by MNIPC. Therefore, the two modules must coexist in the same Mach task.

### 3.2 Transferring Port Rights: SRX and RRX

The port transfer protocols are send-right-transfer (SRX) and receive-right-transfer (RRX). They work in concert with PORTM to assure that the holder of a send right always knows the address of the receiver, and the holder of a receive right always knows the addresses of the senders.

When the port right is being transferred between two machines, and these machines constitute both the receiver and the only sender, then the port transfer protocols are not invoked. But when three or more machines are involved, the transfer protocols are needed to ensure that a consistent view exists between all parties. The transfer protocols achieve this by keeping the port locked (*i.e.*, unavailable for sending messages), until all necessary parties are either informed of the transfer or have rebooted.

Multi-party transfers are initiated and directed by the node holding the right to be transferred. At any point in the transfer protocol, the right is either clearly identified with exactly one node (the old or new node), or with *either* the old or new. In the first case, a reboot of the owner node destroys the right in transit. In the latter case, a reboot of *either* node destroys the right. If the right is a send right, a reboot of the receiver node will also destroy the right. This protocol should be amenable to analysis for correctness with respect to node failure because the state of the right is never strictly dependent on more than two nodes at a time.

Port rights that are transferred using these protocols arrive at their destination and are entered in the database, but they are not fully activated until a Mach message containing a reference to the right is delivered to the machine. When the message arrives, the port manager assumes responsibility for watching the health of the remote senders or receivers, and it informs the port transfer protocols that the message has arrived. Because several messages carrying send rights to the same port may be in transit to a machine, the port transfer protocols have responsibility for the rights until all such messages have arrived. This is why a message identifier is needed in the network header.

The port transfer protocols use the standard *x*-kernel interface for utilizing the transport protocols. However, the interface to the PORTM module is non-standard. This is because the port transfer protocols notify the PORTM module about changes in port states, and these changes are outside the normal sort of data delivery that protocols use.

### 3.3 Reboot Detection: BootId and BootId Control

As described previously, machine reboots define the notion of failure in our system. Reboots are detected by changes in the incarnation number associated with a machine. Unique identifiers for the incarnation number of a node are handled through two protocols: BootId (BID) and BootId Control (BIDCTL). The BootId protocol is tightly integrated with the xRPC protocol subgraph. It is responsible for detecting boot identifier mismatches. Mismatches occur when a message arrives

### XRPC Protocol Stack

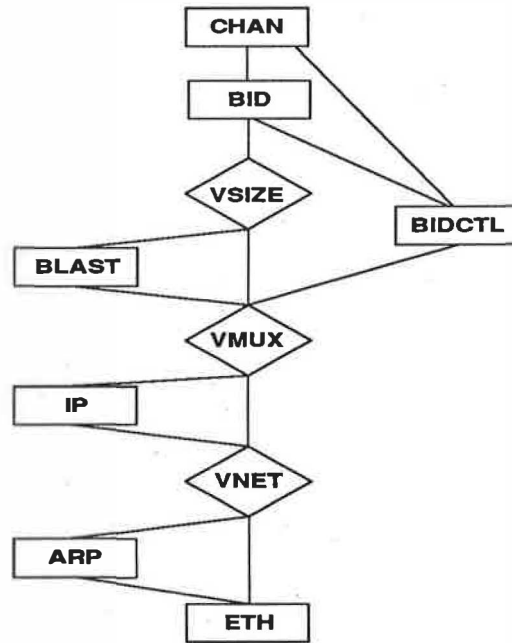


Figure 4: The transport and reliability protocols of Mach NetIPC

with an unexpected boot identifier for either the local or remote machine. The BootId Control protocol handles generation and maintenance of the boot identifier database, probing for changes in boot identifiers, notifying local protocols about reboots, and accepting requests from local protocols to enter or remove a remote machine from the database.

When a node reboots, it changes its incarnation number and broadcasts the new number. Communicants receiving this broadcast request confirmation of the reboot using a nonce to protect against delayed or spurious broadcasts. Upon receiving confirmation, Bootid Control informs interested local protocols of the reboot. Communicants that do not receive the broadcast will learn of the change when they attempt to communicate with the rebooted node, because the BootId protocol will reject traffic that uses the old number and will initiate the confirmation process for the new number. Note that the initial broadcast message accelerates reboot detection for local machines, but the correctness of the algorithm does not depend on it.

Guaranteeing that an incarnation number is not re-used is important to this design. We are using the system time to generate the number, but this is slightly unsatisfactory because clocks can be inaccurate (even “stuck”, especially during the boot procedure). This problem can be ameliorated by storing the previous incarnation id in some form of non-volatile memory, but ideally a true random number generator device would be available.

### 3.4 The RPC Protocol Subgraph: xRPC

Several protocols in a complex configuration comprise the protocol grouping that we call xRPC, the transport service of Mach NetIPC. The graph that we have used during our development and testing is given in Figure 4. The protocols are described in this and the following sections. The transport service is designed to provide efficient delivery in ethernet and inter-network topologies. While it is specially designed to support RPC interactions, it is also appropriate for uni-directional streams.

### 3.4.1 RPC Transport Service: CHAN

The CHAN protocol is the RPC transport protocol that directly underlies Mach NetIPC. It has the desirable properties of:

- Multiple client-server channels (a pair of nodes can have many simultaneous channels between them)
- Acknowledgement of a request is packaged with the reply data
- Duplicate message elimination (“at most once” semantics)
- Data sequencing on a channel matches Mach sequenced message queues
- Minimal state required for each channel
- Message size up to  $2^{32} - 1$  bytes (Note that lower protocols may not support the full size)

The client (or initiating) side supports an *open* operation by assigning a new channel identifier and initializing the state. It also registers interest in the destination with the BootId Control protocol. When a message is sent over the channel, CHAN enters the sending state and expects either an acknowledgement or a reply (data plus ack). If neither arrives in a timely manner, a request for an ack will be sent. If the message is small, it will be retransmitted along with the request for an ack.

The server (or remote) side accepts the first message on a new channel by registering interest in the sender with BootId Control and by initializing its state. The message is delivered “upward”, and the protocol enters the state of waiting for a reply from the server.

When the reply is eventually sent from the server back to the client, the server expects an ack from the client. A symmetric request for ack from the client is carried out if a new request (implicitly acking the reply) is not received in a timely manner. If an explicit ack (with no new request) is received by the server, the channel enters an inactive state.

Inactive channels are persistent objects: they remain available for reuse between the same nodes. This minimizes the overhead of starting up a channel and eliminates the complexity necessary to reliably tear down a channel (cf TCP). Channel objects are destroyed upon notification of a peer rebooting.

### 3.4.2 Virtual Protocols: VSIZE, VMUX, and VNET

Below BootId and CHAN, a series of *virtual protocols* [7] are used to select the appropriate transport protocols, based upon various properties of the message or connection. The first virtual protocol, VSIZE, dynamically directs outgoing messages to either VMUX or BLAST depending upon the size of the message. At connection open time, VSIZE opens both VMUX and BLAST connections and determines the optimal packet size of the VMUX connection. In the graph in Figure 4, the optimal size for VMUX is the Ethernet maximum packet size. VSIZE will send short packets which do not need fragmentation directly to VMUX, avoiding the BLAST protocol. BLAST itself implements a simple blast algorithm with selective retransmission for sending large packets between nodes. This algorithm is tuned for the case where both machines are on the same local net. Note that since BLAST has a maximum packet size it determines the maximum packet size supported by this implementation of the Mach NetIPC (currently 32K bytes).

The second virtual protocol, VMUX, bypasses the IP protocol in cases where the target machine is reachable on the local area network. The advantages of using IP are well-known: nearly a million other hosts are potentially accessible on the Internet. However, the IP service is unnecessary for connections between hosts on the same local network. When a VMUX connection is opened, VMUX attempts to open VNET. If that open succeeds, the target host is accessible on the local area network and all data packets are directed to VNET, bypassing IP. If the VNET *open* fails, an *open* is performed on IP and all data packets are directed to IP.

The third virtual protocol, VNET, isolates the rest of the protocol graph from the existence of multiple physical network links. VNET is configured with (physical network, address resolution

protocol) pairs. VNET is opened using the IP address of the remote host, which it attempts to match with one of the physical networks below it using IP network masks. If it finds a match, VNET opens a connection with the appropriate physical network. In the configuration shown in Figure 4 there is only one physical network (ETH) and one address resolution protocol (ARP).

The graph presented in Figure 4 has three limitations. First it limits the maximum size of a Mach IPC message. This is obviously unacceptable. Second, BLAST lacks any form of congestion control and hence is not well suited to sending large messages over the Internet. Third, it does not provide efficient support in situations where a user sends several non-RPC messages to the same port. All of these problems can be solved by including a stream oriented protocol such as TCP in the graph. We are currently working on a new graph in which TCP will be used for larger messages than BLAST will support, any large message over the Internet, and situations where the Mach IPC protocol has detected that the use of a port would benefit from streaming. Except for the detection of streaming, the addition of TCP should not require any modifications to the higher level protocols.

### 3.5 Locality Issues

All of our code which is concerned with manipulation of Mach IPC data structures (data conversion, port allocation, mach\_msg operations, etc.) resides in a user task, much like the Mach Netmsgserver. There are two problems that derive from this. One is that send-once rights cannot be matched with their corresponding receive rights. This leads to a "ghosting" state in moving receive rights, wherein the old receiver waits for local kernel notification that all send-once rights associated with the port are used. Only when this occurs can it stop forwarding messages sent via send-once rights to the new receiver. A similar situation occurs when a send-once right is moved between nodes twice; in this case the right acquires two global names, both of which will be consumed in sequence when the right is used to send a message. This is an inefficient way to use a right, although such a circumstance is reportedly rare.

When moving a receive right, much of the port state must be retrieved from the microkernel via interface calls, and the queue of messages must be completely transferred to the new node before allowing additional enqueueing. Within the microkernel, the port state could be accessed more quickly, and the queue could be controlled more directly by blocking active senders.

## 4 Performance Results

With all protocols located in user space, the RPC roundtrip time for a 50 byte simple reply/request operation, using a standard Mach kernel running on a 25MHz DecStation 5000 Model 200, is 6.3 milliseconds. Of this time, the overhead due to the Mach NetIPC module alone, is 2.3 milliseconds. This is about twice as high as expected, based on our experiences in implementing other protocols, and we expect to reduce the number to about a millisecond. In contrast, the same test program, when run with the Netmsgserver, has a 30 millisecond roundtrip cost.

When the xRPC and the failure detection protocols reside in the Mach microkernel, the RPC roundtrip time is reduced to 4.8 milliseconds. The time saving over the case where all protocols reside in user space is largely due to avoiding the Mach device interface to the ethernet driver. For longer messages, up to the network maximum transmission unit (MTU), the time is expected to increase linearly. When the MTU is exceeded, the partitioned interface should demonstrate a more pronounced advantage over the "all-user-space" implementation, due to a reduced number of Mach IPC calls across the user/microkernel boundary. For the partitioned configuration, the cost of the crossing to the microkernel to send and receive data from the lower-level protocols is estimated at 16% of the total roundtrip time.

Of the time spent in Mach NetIPC, approximately 10% is due to Mach interface calls to determine the type of reply right being used, to manipulate reference counts for ports, and to request/field Mach notifications. We estimate that another 15% is spent in lock operations; most *x*-kernel operations require acquisition of the master lock to protect common resources, and send rights are protected by locking while messages are in transit through those rights. At least another 20% is estimated to



be due to overhead of the cthreads package: scheduling and activating a thread in the client and server.

The NORMA implementation cites 2.5 milliseconds for RPC roundtrip time [1] for i486 PC's. While it is difficult to compare expected performance between the PC and a DecStation, we expect the DecStation to be at least as fast. For comparison, we note that the NORMA code is entirely resident in the microkernel, and we estimate that .8 milliseconds of our roundtrip cost would be eliminated if our Mach NetIPC code were similarly resident. This still indicates Mach NetIPC would have best case 4.0 millisecond roundtrip time, for 50 byte messages. Our code is not yet fully optimized, and we would expect to be fully comparable with the NORMA implementation within the coming months. This is based on observing that our transport service takes well under 2 milliseconds for 50 byte messages and that the processing required for simple messages is not high.

A detailed timing breakdown for the partitioned configuration is given in Figure 5. Each box shows the transit time used in an RPC between two machines. Where two times are given, the first represents the time used in the "down" direction, and the second represents the "up" direction. The RPC times for a Mach RPC and a CHAN RPC, using 50 bytes messages, are shown at the side.

The time for processing and dispatching an ethernet packet interrupt were obtained by indirect measurement techniques, but the others were measured by using tests that removed one processing layer at a time, running hundreds or thousands of message per test session. The Mach NetIPC time is shown as 1.1 milliseconds for the initiating side, and 1.2 for the replying side. This division is a guess; the server side of most protocols does more work, because it must associate the incoming message with state information for the session. For Mach NetIPC this is a small percentage of the overall time.

## 5 Proposed Mach IPC Changes

Most of our work in achieving an efficient mechanism have been tied to the concept of "fast-path" messages, but the time penalty for avoiding this pathway can be high. Things that cause concern are operations that require touching many bytes of the message, operations like copying, transforming, checking. We made little attempt to optimize these operations, because where they are required they are inherently expensive. These concerns have lead us to suggest three changes to the Mach 3.0 IPC specification: data streams, ordered messages, and untyped data. These changes have been tentatively accepted in the ongoing process of creating a new common Mach interface specification [6]. Note that the software described earlier in this paper implements the original Mach 3.0 IPC definition and does not reflect these changes.

The proposed changes all relate to the Mach message format. The current Mach message format is a flat unordered series of tagged data items. The supported base type tags are: unstructured, bit, boolean, integer\_16, integer\_32, char, byte, integer\_8, real, string, string\_c and several port right tags. The Mach message format also supports one dimensional arrays of these base types and allows any tagged item to be marked as out-of-line data. Normally a message sender creates a message template with tags; each time the template is used for sending, the data to be sent is copied into the template buffer after the appropriate tag. The tags provide enough information to allow the local and distributed Mach IPC implementations to perform the correct actions on embedded ports and out-of-line data segments<sup>1</sup>. The tags also provide enough information to allow the distributed Mach IPC implementation to perform the appropriate conversion on the user data (such as integers) when the data representation of the sending and receiving machine differ.

Avoiding copying is the grail which all network specialists seek. Header additions/deletions, fragmentation/reassembly, and format reconstitution usually result in copying. The x-kernel internal libraries are constructed to help avoid copying, but it has been difficult for us to match this to Mach messages, which must be contiguous or else use the out-of-line option. The first proposed modification to Mach IPC is to add data streaming: a third method of passing user data in a Mach message. As currently defined user data in a Mach message must either be copied into the Mach message template—incurring an extra copy—or passed as out-of-line data, in which case

---

<sup>1</sup> Out-of-line data is "eagerly" copied between nodes, i.e. no cross-node memory objects are created.

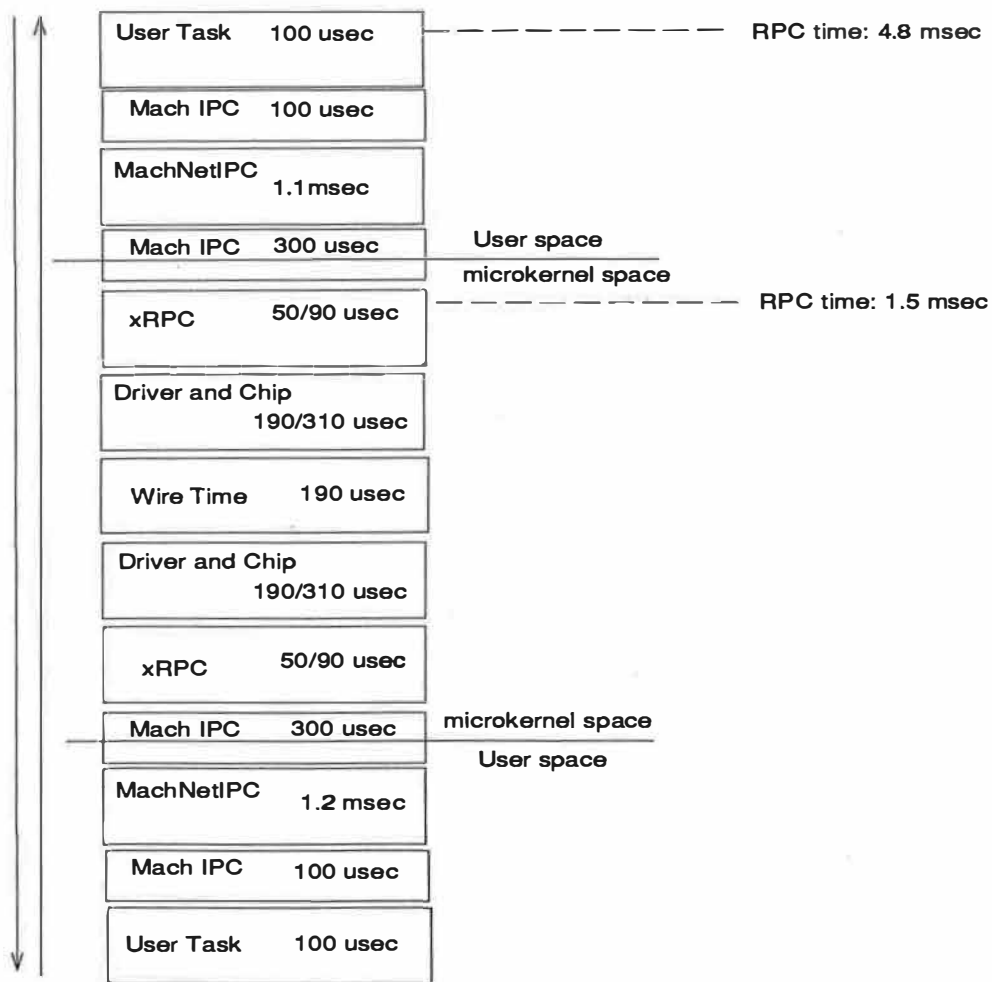


Figure 5: Timing model, protocols split between microkernel and user space

only the data address is written into the message template, but copy-on-write semantics will be used to transfer the data—incurring the copy-on-write overhead. Given that the break-even point for data transferred using copy-on-write is large (on the order of 4000 bytes), we propose that the Mach message format be modified to allow out-of-line data which is eagerly copied by the Mach IPC implementation rather than transferred using copy-on-write.

Another concern has been the expense of transferring ports embedded in a Mach message. The ports in the header, the remote and local ports, are handled through our “fast-path” mechanisms, but others must be located in the message and changed to a network representation. The second proposed modification to Mach IPC is to improve the data organization in the Mach message. The existing Mach message format allows an arbitrary sequence of tags. Ports and out-of-line data objects may appear anywhere in the message mixed in with typical user data such as integers and character strings. Hence, unless the user marks the message as simple, the Mach NetIPC implementation must interpret the entire Mach message, examining each tag for ports and out-of-line data segments. This expensive process must always be done for non-simple messages at the sending node, and it must be repeated at the receiving node. The sending node adds a list of network port descriptors and a list of out-of-line data arrays to each complex message. These must be integrated into the message at the receiving node. This processing can be minimized by ordering the Mach message. In the new proposed format all ports are at the beginning of the message, followed by all out-of-line data segments and streams, followed by user data.

Our final concern is that Mach messages have typed data that must be translated to the target machine architecture if they are transferred between machines with differing architectures. It is our contention that the proper place for the data transformation is at the point where the data will be used. The data typing is a matter of convention between the sender and receiver, and we believe that Mach IPC should be typeless except for ports. Thus, the third proposed modification to Mach IPC is move the responsibility for the marshaling and de-marshaling of user data from the Mach IPC mechanism to the message generator (MIG). This is accomplished by eliminating the base type tags that require conversion (bit, boolean, integer\_16, integer\_32, char, byte, integer\_8, real, string, string.c) and only supporting the type tag *unstructured* and the various port right tags. As currently proposed, the sending MIG stub will be responsible for affixing a NIDL-like [8] architecture tag to data; the tag will be checked by the receiving MIG stub before unbundling the Mach message. This removes data representation from the responsibility of Mach NetIPC and makes it a strictly end-to-end issue. In particular, this change will not require that data items be marshaled when the sender and receiver are on the same machine.

When the effects of these three proposed modifications are combined, the resulting Mach IPC implementation should be significantly simpler and faster overall. Applications can avoid a copy on data items too small to take advantage of copy-on-write. The need to repeatedly scan messages looking for ports is eliminated and it is also much easier to tell if a specific message does or does not contain additional ports. User data is now treated as an undifferentiated byte stream which is neither examined nor modified by either the local IPC or Mach NetIPC implementation. The elimination of user type tags significantly simplifies the Mach NetIPC implementation and eliminates one copy in cases where data is sent between machines with different data representations. Finally the formatting of user data is now completely the responsibility of MIG which will make extending or changing the currently restricted user data representation format much simpler.

## 6 Conclusions and Further Work

This work demonstrates that modular protocol design can be used to decompose the Mach IPC mechanism into efficient protocol units. It is possible to operate over either LAN's or internetwork configurations without any additional coding. Placing the transport protocols into the Mach kernel yields a significant performance benefit, a fact that has been previously noted with respect to the Mach 2.5 kernel with TCP/IP [1]. The timing breakdown indicates that moving the port manipulation functions into the Mach kernel would make a large difference.

This implementation does not define the legal participants of a communicating network. It

is likely that administrators would like to limit the communication to selected node lists and/or subnets. This could be achieved by adding an authorization protocol to be used in conjunction with the port transfer protocol.

Cryptographic methods for assuring message integrity and privacy are being developed as generic protocol components in separate research. Digital signatures modules developed as part of this work could be used to support the security of the authorization protocol.

For machines with special-purpose communication hardware, our approach is still valid. The xRPC stack could be replaced by any code that obeys the x-kernel interface conventions and utilizes the message library and thread locking conventions. The overhead of the x-kernel interface operations is already low, and it should be an acceptable cost for communication-intensive operations that work through Mach IPC.

Finally, an x-kernel that distributes processing among tightly coupled processors is an open issue. The current implementation is MP-safe in the trivial sense that if the master lock acquisition is honored, the x-kernel threads can be distributed to separate processors. To achieve a finer processing granularity, much of the locking must be reworked. This reworking would have to include the issue of locks within the Mach NetIPC database, as well.

## 7 Acknowledgement

Richard Schroepel did the detailed timing breakdowns and improved the ethernet device driver, making it possible to get accurate latency measurements for RPC's.

## References

- [1] J. S. Barrera. A fast Mach network IPC implementation. In *Proceedings of the Usenix Mach Symposium*, pages 1-12, 2560 Ninth Street, Suite 215, Berkeley CA 94710, November 1991. Usenix Association.
- [2] A. Birrell and B. Nelson. Implementing remote procedure calls. *ACM Trans. Comput. Syst.*, 2(1):39-59, Feb. 1984.
- [3] N. Hutchinson, L. Peterson, S. O'Malley, E. Menze, and H. Orman. *The x-Kernel Programmer's Manual (version 3.2)*. Computer Science Department, University of Arizona, Tucson, Arizona, January 1992.
- [4] N. C. Hutchinson and L. L. Peterson. The x-Kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering*, 17(1):64-76, Jan. 1991.
- [5] D. P. Julin. Network server protocol specification (draft). Computer Science Department, CMU, August 1989.
- [6] K. Loepere. Mach 3 kernel principles, draft proposed specification. Open Software Foundation, July 28 1992.
- [7] S. O'Malley and L. Peterson. A dynamic network architecture. *ACM Transactions on Computer Systems*, 10(2):110-143, May 1992.
- [8] Network computing architecture, version 2.0 specifications. Open Software Foundation, September 25 1992.
- [9] J. K. Ousterhout *et al.* The Sprite network operating system. *IEEE Computer*, 1988.
- [10] R. Schlichting and F. Schneider. Fail-stop processors: An approach to designing fault tolerant computing systems. *ACM Trans. Comput. Syst.*, 1(3):222-238, Aug. 1983.

# Port Buffers : A Mach IPC Optimization for Handling Large Volumes of Small Messages

Kenneth W. Koontz

*The Johns Hopkins University  
Applied Physics Laboratory  
Laurel, MD 20723\**

*Kenneth\_Koontz@jhuapl.edu*

## Abstract

An extension to the Mach kernel's communications mechanism is described which optimizes the handling of large volumes of small messages. *Port buffers* allow a task to pack several short Mach messages into a larger message that is sent atomically to a receiving task. A staleness feature allows the sender to guarantee that messages placed in the buffer are sent within some desired maximum time period. Though initially designed to improve network utilization between Mach hosts connected by Ethernet, it was also found to improve local transfer rates by reducing the number of kernel calls and context switches. Port buffers may also be useful for increasing the utilization of high-speed networks such as those found in non-shared memory parallel architectures and future distributed systems.

This paper describes the model, implementation, and performance of our first port buffers package. The implementation supports the full range of Mach message formats as well as providing multi-threaded support for senders and receivers. The initial implementation improves remote I/O message delivery rates by more than an order of magnitude; local transfer rates are also improved by a factor of three. Currently, this mechanism is provided through a library that is linked into a task desiring such optimization. Most functions can be performed well in user space, eliminating the need for building this optimization into the kernel.

## 1 Introduction

Many optimizations have been made over the past years to improve the Mach messaging system. The transfer of very large messages via out-of-line pointers, well integrated with Mach's virtual memory management, has been a key feature of Mach since its early macro-kernel days [1]. To improve client/server performance in the microkernel environment, many improvements were made to efficiently handle Remote Procedure Calls (RPCs) in Mach 3.0 [2]. To improve communications between nodes in a non-shared distributed memory architecture, fast in-kernel messaging was developed to reduce context switching and simplify the handling of proxy ports [3].

Yet the need to optimize handling of large volumes of small messages has been overlooked. Many event-driven systems use small messages (< 100 bytes) to exchange data and affect state changes in the tasks and threads that comprise the application. While this design phi-

---

\* This research was supported by the Defense Advanced Research Projects Agency (DARPA) and the Aegis Program Office (PMS-400) and monitored by the Space and Naval Warfare Systems Command under Contract N00039-91-C-0001.

losophy is an effective approach to software organization, the large body of small messages can cause significant amounts of context switching on a single processor or symmetric multiprocessor. But when run over a collection of distributed hosts connected via a network, high message setup costs can reduce the effective throughput rates to an unsuitable level.

This problem was brought to light while porting a multi-platform radar tracking and correlation system to a distributed system of Mach workstations. The event-driven design of this application can produce on the order of 10,000 small messages per second with large track loads. But even minute track loads can produce message rates that easily exceed the capabilities of the standard network message server [4]. Due to the server's use of standard Unix sockets and TCP/IP, asynchronous remote port message throughput was limited to approximately 130 messages per second, regardless of message size [5]. Since setup time, not wire time, predominates with small messages, simply replacing Ethernet with a faster network (e.g., FDDI) was not a solution.

To improve our effective utilization of network resources, the generalized concept of buffering was applied to Mach's model of inter-process communications (IPC). A new object, the *port buffer* [6], was defined which can be assigned to the send and receive rights of a port. A port buffer augments the port queue already present in Mach and allows a number of small messages to be packed together and sent as a single message. A port buffer can be created with an associated *maximum staleness* which specifies the maximum time that a message should sit in the buffer before it is sent. This allows messages to be delivered in a timely manner but with the advantage of increased throughput if multiple messages are present. The facility is somewhat analogous to ISIS' *pg\_buffer* command, though port buffers operate on a Mach port as opposed to an ISIS group [7, 8].

The implementation has been designed to be very general purpose, simple, yet feature laden. It is treated as an extension to the Mach port-based communications system. While developed specifically to improve remote port throughput, it was also found to improve local transfer rates as well. The initial port buffers implementation has the following capabilities:

- Insensitive to port location (port communications can be local on same host or remote on different host).
- Insensitive to type of remote port facility (network message server or in-kernel fast messaging).
- Buffering on both port send-right and receive-right.
- Multiple reader and writer threads supported.
- Handles simple data (in-line), complex data (out-of-line and port rights), or any mix of multiple data types in a message.
- Any number of buffers allowed per task (limited only by virtual memory system limits).
- Buffer sent when no room for next message (full), when optionally specified maximum staleness period expires, or when forced by user.
- Supports dynamic buffer creation, destruction, and monitoring.
- Implemented in user space (no kernel modifications required).
- Supports Mach 3.0 message calls and formats.

This paper describes the model, implementation, and performance of our initial port buffers package. The model of Mach port-based communications, augmented with the port buffer object, is first described. The “pseudo-kernel” operations on port buffers are then presented. Details of the internal workings of port buffers are then briefly described including the Buffer Control Block (BCB), packing format, and timer thread. Performance measurements of local and remote communications with and without port buffers are then presented. Finally, we conclude with a look at limitations and possible future improvements of the port buffers approach.

## 2 Augmenting Mach's IPC Model

Mach's IPC model revolves around the sending and receiving of messages by tasks. Messages are sent through ports; two tasks desiring communications are connected by a port. To send a message, a task must hold send rights to that port. Likewise, to receive a message, a task must hold receive rights to that port. Each task holds a number of ports in its port name space which is accessible by all threads running within that task. Each port name (normally represented by an integer) is unique for each task but is not unique within the system; the Mach kernel performs port name associations between different tasks. A port can have multiple senders (or send rights) but only one receiver (or receive right).

Associated with the port at the receive right is a message queue maintained by the kernel. The queue allows senders to send messages to the receiver in an asynchronous, non-blocking manner. The sender will be blocked only when there are no more positions open in the queue. While the size of each message in the queue is unlimited (especially with the use of out-of-line data), the number of messages that can be queued is normally quite small (the default value of five messages can be extended to a system defined limit via a kernel call). A much more complete description of these basic principles is described in Reference 9.

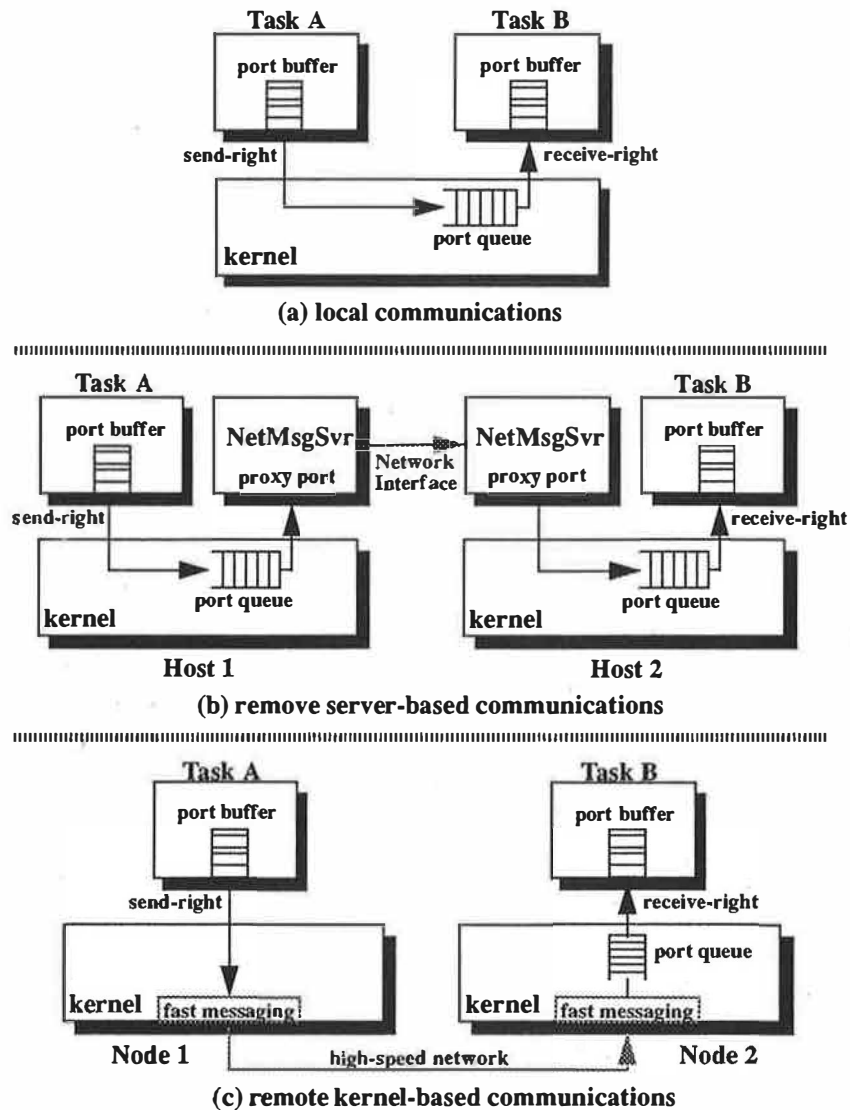
Port-based communications can be extended across hosts in a distributed system or nodes in a non-shared memory parallel architecture to allow tasks running on different processors to communicate via the same mechanism. This facility is one of the most powerful features of Mach, allowing tasks to communicate with each other without having to know their exact locations; the sender only needs to acquire a send-right to communicate with a receiver. In distributed systems of workstations, port rights and messaging are extended across host boundaries using a Network Message Server [10]. In a no remote memory access (NORMA) parallel architecture such as an Intel Touchstone Sigma / Paragon, fast in-kernel messaging [3] supports remote port mapping and communications.

While these facilities allow ports to be extended over remote hosts, the hardware / software interface between servers (or kernel instances for NORMA implementations) can impose restrictions on the rate of data that can be sent between two tasks. For each message sent from one task to another, the server (or kernel) must setup and send a message over the network. When the messages are large, the setup overhead is normally small relative to the wire time of the message on the network hardware. But when messages are small, the setup time can dominate and severely restrict the message throughput rate and network bandwidth utilization.

To help increase the amount of data that is sent between the two tasks, a port buffer can be created at each port right. When a task sends a message to a buffered port, it does not send it immediately; the message is temporarily packed in a memory buffer local to the sending task. When enough messages are available, an *atomic send* is performed on the buffer, i.e. its contents are sent as a single large message with one `mach_msg` send. When receiving messages via a

buffered port, messages are unpacked from the buffer one-by-one; when the buffer is empty, an *atomic receive* is performed (i.e. a set of packed messages are received with a single `mach_msg` receive) to fill the buffer before the first message is unpacked and returned. The port queue continues to provide flow control; the sender is only blocked during an atomic send when the queue is full. But now the queue is holding collections of packed messages en route between buffers. This mechanism, in keeping with the spirit of Mach IPC, is transparent to the location of the ports. Buffers can be used between tasks running on the same host (Figure 1a), tasks running on different hosts linked via network message servers (Figure 1b), or tasks running on different nodes of a non-shared memory multiprocessor using fast in-kernel remote communications (Figure 1c).

**Figure 1 : Communications Between Two Tasks Using Port Buffers**



While atomic receives are always performed whenever the buffer is empty, atomic sends occur under one of three conditions. If the buffer is full, i.e. the next message to pack in the buffer will not fit in the remaining space, the buffer's contents will be sent. In some cases, waiting until the buffer is full is undesirable; if the flow of data is not constant or suddenly falls, messages may sit in the buffer for an undesirable length of time. To overcome this problem, the sender's



buffer can be created with a specified *maximum staleness period*. When the first message is written into an empty buffer, a timer is started to tick-down the maximum staleness period. If the timer expires before the buffer is full, then an atomic send is performed. But if the buffer is filled before the timer expires, the timer is aborted and the contents of the buffer are sent early. An atomic send can also be performed by the user forcing a send before either the buffer is full or staleness has been reached.

With the port buffers in place, other Mach IPC mechanisms continue to work as usual. Mach's guarantees for ordering of delivered messages are still maintained since they only guarantee that messages sent from one task to another task are received in the same order as they are sent. A single receive-right must still be observed but multiple buffered send-rights are allowed. Any type of Mach message can be sent through the buffers including out-of-line data and port rights. Homogeneous or heterogeneous environments are still supported with data reformatting performed via the network servers between the buffers. And as always, messages can be sent and received on standard, unbuffered ports via `mach_msg`.

### 3 Pseudo-Kernel Calls

The initial implementation defines a set of *pseudo-kernel calls* of primitive operations to manipulate port buffer objects. While the calls look like Mach kernel calls, they are implemented as an object module and execute entirely within user space. The implementation required no changes or special hooks into the kernel; standard kernel calls and message formats were used throughout. This approach is highly portable over different versions of Mach 3.0 (CMU vs. OSF), different remote messaging facilities (server-based vs. in-kernel based), and different hardware (collections of workstations interconnected via Ethernet or FDDI vs. NORMA architectures with fast router-based networks).

Seven calls are currently defined to implement the buffer primitives:

**port\_buffer\_create (port, size, staleness):** creates a port buffer in the current task of a particular size (bytes) and with the given staleness (msecs) assigned to the specified port name. If the port is a send-right and staleness is specified, a timer thread is also created to manage the staleness timing. If `NO_STALENESS` is specified, the buffer will only be sent when the port is full or if forced by a `port_buffer_force_send`. A value of `staleness = 0` is also allowed which will send the message as soon as it is packed in the buffer (effectively disabling buffering). Currently, the size of the buffer and its staleness are constant for the life of the buffer.

**port\_buffer\_destroy (port):** destroys a port buffer, its timer thread (if a send-right), and any memory consumed for packing messages or maintaining the state of the buffer.

**port\_buffer\_get\_status (port, &status):** returns the current state of the buffer. State can be short-term information (what is currently in the buffer) or long-term (what has the buffer been doing since it was created). Current status information includes:

- type of buffer (send or receive).
- buffer size (specified during create).
- maximum staleness (specified during create).
- number of messages currently packed in the buffer (short-term).

- number of bytes used in the buffer (short-term).
- number of bytes left in the buffer (short-term).
- number of mach\_msg send or receive errors (long-term).
- number of buffers sent/received through the port (long-term).
- number of buffers sent due to staleness exceeded (long-term).

**port\_buffer\_force\_send (port):** flushes the contents of a buffer before the staleness timer expires. Useful for flushing the buffer before it is destroyed, sending the contents due to some external condition (pseudo-priority), or to support a user-specified staleness feature (in the event the internal staleness feature is not responsive enough).

**mach\_msg\_send\_buffered (&msg\_hdr):** sends a Mach message but via the buffer. If room is available, packs the message into the buffer. If the first message, initializes staleness timing (if specified). If no room, performs an atomic send to make room. Also provides all concurrency control to allow access to the buffer by multiple writing threads in the task as well as the timer thread.

**mach\_msg\_rcv\_buffered (&msg\_hdr, timeout):** receives a Mach message through a port buffer. If the buffer is empty, performs an atomic receive to fill the buffer with messages from a sender. If not empty, unpacks and returns the next available message. If no messages are available and a timeout is specified, tries to perform an atomic receive with the indicated timeout. Also provides all concurrency control to allow access to the buffer by multiple reading threads.

**port\_buffer\_module\_init:** initializes the port buffers package in the task. This call must be performed shortly after the task's main thread is started before any buffers can be created and used.

## 4 Implementation Overview

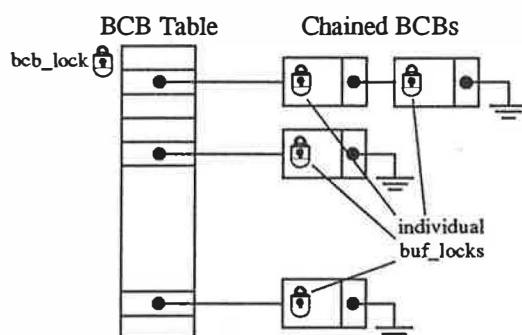
The buffer module allows a task to build and use multiple buffers on different ports, be they ports with send-rights or receive-rights. Each buffer has an associated Buffer Control Block (BCB) that is used to maintain the state of one port buffer. Each BCB has associated with it a buffer memory and, if the port is a send-right, a timer thread. Multiple threads within a task can access different BCBs concurrently. Locks and condition variables within the BCBs control the access of the threads to the buffer memory.

This section takes a brief look at the internal workings of the buffer module. The BCB and the BCB Table are first described. Then, the method used to pack multiple Mach messages in a buffer is presented. Finally, the implementation of the timer thread is described.

### 4.1 Buffer Control Blocks (BCBs)

The Buffer Control Block (BCB) maintains all information and state about a port buffer, its buffer memory, and associated timer thread (if send-right). All BCBs within a task are maintained in the BCB Table. The BCB Table is implemented as a simple chained hash table which uses the port name as the key (see Figure 2). Since each port name in a task is unique, there can be no collisions for the same key. To reduce search time, BCBs on the same chain are maintained in sorted order by increasing port name. This data structure provides extremely efficient search access; most BCBs are located within the first or second attempt. An unlimited number of buffers can be created and used subject only to virtual memory availability.

**Figure 2 : Organization of BCB Table Data Structure**



Associated with the BCB Table is a mutex lock, the `bcb_lock`. This lock is required to support multi-threaded users. It prevents one thread from inserting a new BCB into the BCB Table for a newly created port buffer while another thread is searching for or destroying a second BCB. But it also limits to one the number of threads that can concurrently search through the table. If a second thread wants to search for a BCB, it must wait until the mutex is unlocked. However, once a BCB is found, a second lock associated with a given buffer's memory (the `buf_lock`) is used to prevent concurrent thread access to the same buffer. Though the single BCB Table mutex limits the number of searching threads to one, the separate buffer locks allow concurrent access to individual BCBs by many threads (limited only to the number of active BCBs available).

Each BCB contains information on a different port buffer maintained by a task. This structure is fairly large and is used to keep track of the state of the buffer memory, to initialize and control the timer thread, and to synchronize multiple threads attempting to access the buffer simultaneously. Note that the BCB and BCB Table are internal structures within the port buffer module. They are not viewable or modifiable directly by the user. The pseudo-kernel call `port_buffer_get_status` will return some viewable parameters in the BCB. All other calls modify the BCB and BCB Table in some (sometimes complex) ways

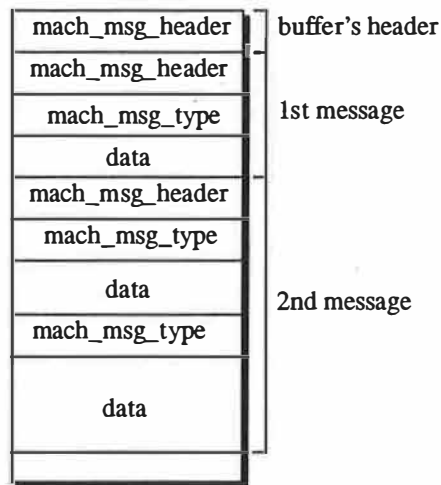
## 4.2 Packing Format

Messages are buffered by condensing multiple Mach messages into a single Mach message. This turns out to be more difficult than it sounds. A Mach message contains a header optionally followed by data (a simple message may only contain a header). The message data consists of one or more type and data fields. Each different type of data (e.g. integers, floats, characters, strings, port rights) must be preceded by its associated type declaration. If data is sent in-line, the data items are contained directly in the message. Optionally, the data can be sent out-of-line to optimize the transfer of large blocks of data using Mach's virtual memory system; in this case, a pointer to the actual data follows the type declaration.

The buffer memory is organized as a Mach message with the header contained in the first six words. This allows the contents of the buffer to be sent and received atomically with a single `mach_msg` call. However, the messages to pack are also Mach messages which contain their own headers. But multiple headers are not allowed; Figure 3 illustrates a packing format that cannot be used.

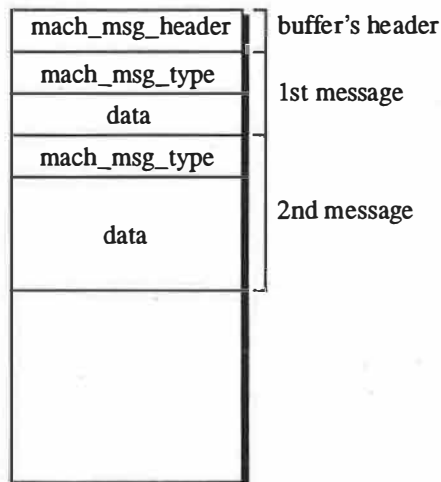
To overcome this problem, there are two possible packing formats that could be followed. The first format allows only one type-data entry per message (see Figure 4). This may be

**Figure 3 : Unrealistic Packing Format w/Multiple Mach Headers**



useful in limited applications but it cannot support the full range of Mach messages formats. In addition, useful information residing within the original message's header (e.g. `msg_id`) is lost. It is up to the user to provide this information (redundantly) within the data (but without type information). This makes this format very specific to the user's data format, not general-purpose, and not very useful in heterogeneous environments.

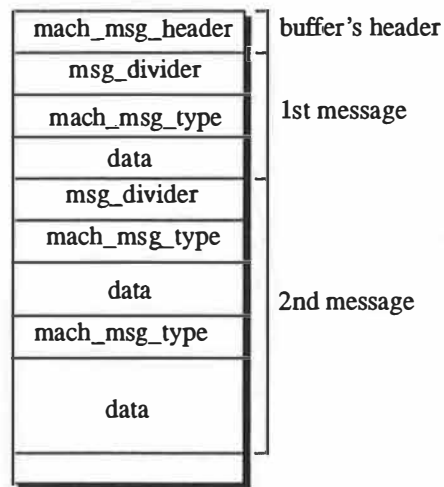
**Figure 4 : Limited Packing Format Using Single Data Types**



The second format does not restrict the message to a single data type but allows any number of data types (including zero) to be packed. This format uses *message dividers* to separate the packed Mach messages and to hold the non-redundant header information required to reconstitute the original headers of the packed messages. A divider consists of a Mach message type with name `MACH_MSG_TYPE_INT32` followed by two unsigned 32-bit integers. The first integer contains the `msg_id` field of the message's header; the second integer contains the `msg_size` field of the header. While the size could be computed from other types within the message, providing it in the divider simplifies unpacking. Figure 5 illustrates the packed format currently used by the buffer module.

The single buffer header which appears as the first six long words in buffer memory are used for a variety of purposes. The port fields (`msg_local_port` and `msg_remote_port`)

**Figure 5 : Flexible Packing Format Using Message Dividers**



specify the source and destination ports (if send, or visa-versa if receive); each message sent through the buffer has these ports in common so they can be specified once in the buffer's header. The `msg_h_size` field contains the total number of bytes in the buffer including the buffer's header, all dividers, and all message types and data. The `msg_h_id` field is used to hold the number of messages packed in the buffer; while the unpacker could determine this count by the number of dividers present, using a message counter again simplifies unpacking at the receiver. The `msg_h_bits` field is set to correctly pass the header port rights for an asynchronous transfer and to indicate the presence of out-of-line or ports information in a packed message. If a message which is packed has the *complex bit* set in its `msg_h_bits` field, then the complex bit in the buffer's header will be set. This allows out-of-line data and port rights data to be packed along with the more conventional in-line data.

While dividers provide a great deal of flexibility, they are not without cost; each divider requires three additional 32-bit words (12 Bytes). While this is more overhead than the single-type-per-message format, it only stores the relevant fields from the original message header that must be used for unpacking at the receiver. Storing the full header would require six 32-bit integers and would include redundant data (e.g. the local and remote ports). Thus, using dividers is an optimized trade-off between the desire to support full Mach message type capabilities and limiting overhead in the packing format.

### 4.3 The Timer Thread

Each BCB corresponding to a port's send-right has an associated *timer thread*. This thread is responsible for limiting the staleness of messages placed in the buffer to the maximum staleness period specified by the user when the port buffer was created. It is also responsible for sending the buffer to the port. Giving the job of sending the buffer to the timer thread simplifies the design considerably; the locks and conditions controlling concurrency only need to deal with a single reader / multiple writers problem.

The timer thread is created during a `port_buffer_create` call. It is currently implemented as a wired-down, detached Cthread. The use of Cthreads was preferred since they are easier to use and are more portable than raw Mach threads (provided the target host also supports Cthreads). But they cannot be directly suspended, resumed, or destroyed. However, if a Cthread is wired-down to a single, unchanging Mach thread using `cthread_wire`, the thread control port of the

Mach thread can then be obtained with `mach_thread_self()`. The timer thread performs these two functions immediately after being forked and detached, and stores its Mach thread\_id in the BCB. This then allows the timer thread to be terminated by another thread when a `port_buffer_destroy` is required.

Once created, the timer thread sits idle waiting for the first message to be written into the buffer. When this occurs, it begins to time the `max_staleness` of this message. The timing function is currently implemented using a `mach_msg` receive call on a *timer\_port* specific to each BCB. If the receive operation fails due to a timeout, the maximum staleness of the first message has been reached. The timer thread will then grab the buffer, send its contents to the port, reinitialize the buffer to empty, and wait for the next first-message-written condition.

A timer thread can be interrupted during its staleness timing to send the contents of the buffer early. This occurs when either the space remaining in the buffer is too small to pack the next message or a `port_buffer_force_send` call is performed to force the contents of the buffer to be sent immediately. Interruption is performed by having the *packing thread* (the one calling `mach_msg_send_buffered` or `port_buffer_force_send`) send a null message to the timer thread on the `timer_port`. If `NO_STALENESS` timing is specified for `port_buffer_create`, the timer thread is still created. However, the thread will not perform a receive with timeout operation on the `timer_port`, only a receive. The timer thread will then only send the buffer after it receives a message on the `timer_port` (either by a buffer full or a force send condition).

Currently, a maximum staleness of zero will cause the timer thread to attempt a receive on the `timer_port` with a timeout value of zero milliseconds. This still may not cause the buffer to be sent immediately since the timer thread must make a quick `mach_msg` call and grab the buffer's lock before the buffer can be sent. If in a system it is desirable to send the bulk of data through a port buffer but, on occasion, a high-priority message must be sent quickly, several options could be used:

- Pack the message and then perform a forced send. This will get the message to the receiver sooner than waiting for the staleness timer to elapse. But the message may still have to wait in line behind other messages in the buffer (and possibly multiple buffers in the port queue) before it can be processed at the receiver.
- Or use two ports: one buffered, one unbuffered. Use a standard `mach_msg` call to send the unbuffered data. At the receiving end, assign two different threads at different priorities to the different ports to handle the unbuffered, priority message in a more timely manner.

## 5 Performance Results

To measure the effectiveness of port buffers, a unidirectional stream test was implemented with and without buffers. This test involved two simple tasks: a sender and a receiver. The receiver runs in the background, always ready to receive a message. The sender is run on the same host (node) or a different host (node) to measure local or remote throughput rates. The size and number of messages to send are specified through arguments to the sender. The sender times how long it takes (from its point of view) from before the first message is sent to after it has sent the last message. Provided that far more messages are sent between sender and receiver than can be in-transit in the I/O pipeline, this test gives a good approximation of sustained through-

put between two tasks. All tests were performed using CMU's MK75 release of the Mach 3.0 kernel for i386/486 machines. Two Dell 450 DEs (50 MHz i486), connected via Ethernet with 3Com 3C503 interfaces, were used as the hosts for these tests. Remote port mappings and transfers were performed with CMU's Network Message Server [10] modified to handle Mach 3.0 messages (netmsgserver\_unix).

Transfer rates were measured using senders and receivers with and without port buffers. In each case, the size of the message was varied to obtain performance plots of message size vs. throughput. When port buffers were used, buffers were sized at 4144 bytes with a staleness of 10 msecs. Several different configurations were tested:

- Remote in-line transfers with and without buffers.
- Local in-line transfers with and without buffers.
- Local out-of-line transfers with and without buffers.

Figures 6-9 present log-log plots of the measured performance. Both message rate (msgs/sec) and throughput rate (Kbytes/sec) are plotted.

Figure 6: Remote Message Rate (msgs/sec)

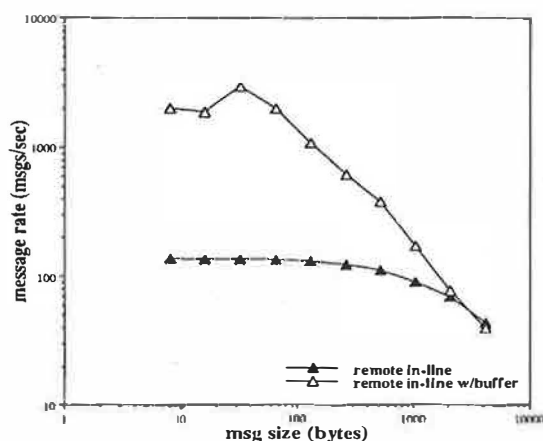


Figure 7: Remote Throughput Rate (Kbytes/sec)

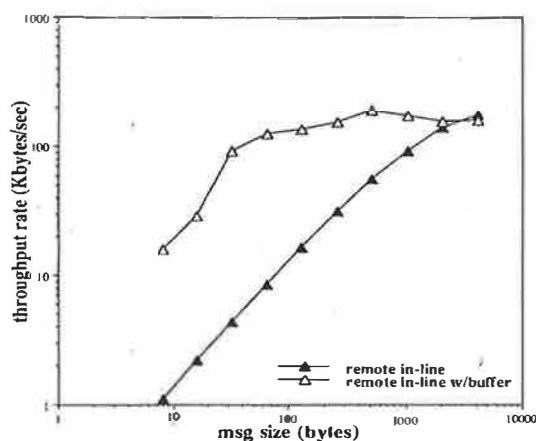


Figure 6 clearly shows more than an order of magnitude improvement in the remote message rate for small messages (< 128 bytes). With buffers, we can expect a maximum small message rate of approximately 2,000 msgs/sec (vs. 130 msgs/sec without buffers). If we look at the throughput graph (Figure 7), we see that the maximum throughput recorded with large messages was around 200 Kbytes/sec (this is the limit of our 3C503 network boards present in the system). Buffers were able to keep the throughput high even with small messages of 32 bytes.

What is surprising is the improvement buffers can make for local transfers. While not as dramatic, in-line local transfers showed a 3x improvement with buffers for small messages while out-of-line transfers showed a 1.5x gain (Figure 8 and 9). But note the faster roll-off of in-line w/buffers vs. without; this is due to the extra copying required to transfer in-line data into and out of a buffer. While the cross-over point between in-line and out-of-line is around 2 Kbytes without buffers, it is around 200 bytes when buffers are used (in this implementation).

Figure 8: Local Message Rate (msgs/sec)

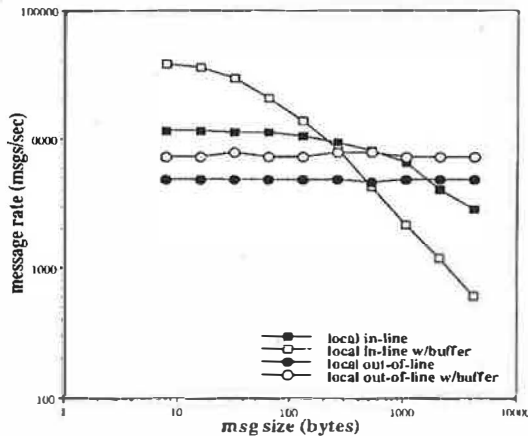
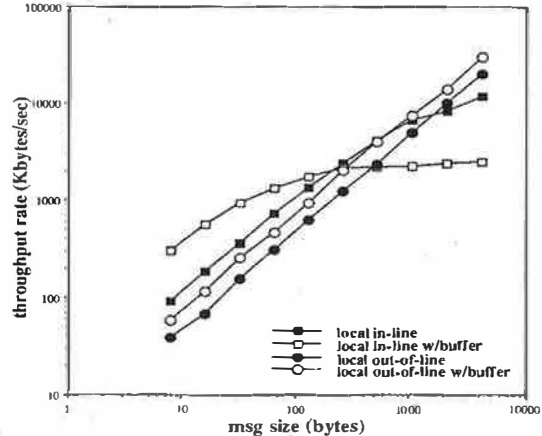


Figure 9: Local Throughput Rate (Kbytes/sec)



## 6 Current Limitations, New Features, and Future Work

The initial implementation has demonstrated the port buffers concept as practical to improve the performance of Mach IPC when moving around large quantities of small messages. During its development, features were kept to a minimum to “keep it simple”. Now that the initial implementation is operational, there are a number of things that probably should have been done differently or features we wish we had added. This section briefly describes some of the current limitations and new features to be addressed in a future implementation.

The staleness timing resolution may be too coarse for some applications. The current implementation is tied to the frequency of the system clock which is typically 100 Hz. With a minimum staleness period specified, a message may experience up to 10 msec additional latency due to the buffer’s presence. In addition, only 100 atomic sends per second may be performed between the sender and receiver (provided they are all due to staleness). Real-time applications may require lower latencies (e.g. 500-2000  $\mu$ sec); faster networks may also show improved performance when finer staleness values are used (as opposed to using bigger buffers). New kernel features such as OSF’s proposed clock facility [11] may prove quite useful in providing finer staleness timing.

A `timer_thread` per send-right buffer was easy to implement but is overkill. If too many port buffers are created, additional scheduling overhead may affect performance. A single scheduling thread could be used instead to time the staleness expiration of many buffers. Packing threads could be designated to perform atomic sends when the buffer is full or forced. But additional *messenger threads* would be needed to provide asynchronous atomic sends once staleness had been reached. A small group of messenger threads could be used to handle a larger number of buffers, but some work is needed to determine the “optimum” ratio (if there is one). Additional messenger threads could be created as buffers are created or when needed; otherwise, an additional waiting period may be imposed after staleness has expired if a messenger was not available to perform the atomic send.

Real-time applications may require more features than just a higher resolution staleness. Priorities between timer and packing threads need to be addressed. Currently, timer threads execute at the same priority as all packing threads. But timer threads need to be at a higher priority than packing threads to allow quick access to the buffer for atomic sends. Buffers might



also we “wired” to improve response time, i.e. disable page-outs of memory pages holding the buffer code, buffer memory, and BCB information.

The current implementation also suffers from a quick-and-dirty interface for message sending and receiving; a better integration with the kernel and the `mach_msg` call is needed. Yet it is desirable to keep most (if not all) of the functionality outside of the kernel. Currently, one cannot mix standard `mach_msg` sends and atomic sends to a buffered port (though this hasn't been a problem since buffered and unbuffered ports are normally segregated in our applications). A simple fix would be to make a generic `mach_msg_buffered` call that would default to `mach_msg` if a BCB record was not available for this port. A better approach would be to modify the `mach_msg` wrapper so that it knows about and handles both buffered and unbuffered ports. This would be the preferred approach if port buffers are offered in new releases of Mach.

There are also lots of little changes that would make the package “better”. Sequence numbers need doctoring on the receive-right. More types of status information should be available. Buffer size and staleness values should be modifiable without having to destroy and create a new buffer. Automatic resizing should be performed so that buffers on the send and receive right are of a compatible size and enlarged to temporarily hold messages larger than the buffers. And port buffers will not currently work on a port that has both send and receive rights.

Future work will also focus on porting the package to other implementations of Mach on different platforms. This will verify the portability of the approach (i.e. a user-space IPC optimization with no kernel changes required) and also provide us with performance figures on different network hardware, protocols, and processors. Of particular interest is the performance of port buffers on the Intel Paragon's fast network backplane. Benchmarks will also be made on the current PC hardware but using a different variant of the Mach 3.0 kernel provided in OSF-1/AD.

Theoretical work is also required to determine the “best” buffer size and staleness period for a system. These two simple variables are highly intertwined with system capabilities and performance in diverse areas such as device interfaces, software protocols, network transmission rates, context switch times, and general processor / network performance. They are also very related to the communications behavior of tasks in an application. Much more work is necessary to develop a model to assist users in selecting the best parameters for a particular system.

## 7 Conclusions

Port buffers can help improve the performance of Mach port-based communications, especially when the message environment is dominated with many small messages. While they were primarily developed to improve remote communications between hosts connected by Ethernet, they also optimize local transfers as well. Port buffers may also be beneficial in improving the utilization of high-speed networks found in non-shared memory parallel architectures such as Intel's Paragon or tomorrow's distributed systems based on gigabit-rate networks.

The port buffer object was inserted into the Mach IPC model with little difficulty, attesting again to the expandability provided by the microkernel. Only 1,000 lines of ‘C’ source code was required to handle a special case of asynchronous message traffic. This out-of-kernel optimization executes entirely in user space, making use of current kernel calls and well established servers to perform its job. Additional kernel facilities such as high-resolution clocks may help improve the performance of future implementations.

The source code to the port buffers package is freely available via anonymous ftp from *aardvark.jhuapl.edu*. Please observe all rights, permits, and disclaimers present in the source code files. A postscript version of this paper is also available from the same host.

### List of References

1. R. F. Rashid, *Threads of a New System*, Unix Review, August 1986, pp.37-49.
2. R. Draves, *A Revised IPC Interface*, Mach Usenix Symposium, October 1990.
3. J. S. Barrera, *A Fast Mach Network IPC Implementation*, Mach Usenix Symposium, October 1991.
4. K. W. Koontz, *Throughput Rates of the HiPer-D Correlator and Tracker (HCT) Transport System based on Mach 2.6 Inter-Process Communications (IPC)*, JHU/APL, F2D-91-2-086, December 4, 1991.
5. J. M. Pierce, *HCT (HiPer-D Correlator and Tracker) Transport System, Analysis of Mach 2.6 Based Implementation*, JHU/APL, F2D-92-2-094, December 18, 1991.
6. K. W. Koontz, *Extending Mach Port-based Communications with Port Buffers*, JHU/APL, F2D-92-2-082, July 8, 1992.
7. K. Birman, K. Marzullo, *ISIS and the META Project*, Sun Technology, Summer 1989, pp.90-104.
8. K. Birman et al, *The ISIS Distributed Toolkit, Version 3.0, Reference Manual*, ISIS Distributed Systems, 1992.
9. K. Loeper, *Mach 3 Kernel Principles*, OSF/CMU, January 7, 1992.
10. Mach Networking Group, *Network Server Design*, CMU, August 31, 1989.
11. K. Loeper, *Mach 3 Kernel Interfaces*, OSF/CMU, Unification: July 28, 1992.

# Using the Mach Communication Primitives in X11

*Michael Ginsberg and Robert V. Baron and Brian N. Bershad*

*mikegins@microsoft.com, rvb@cs.cmu.edu, bershad@cs.cmu.edu*

*School of Computer Science  
Carnegie Mellon University  
5000 Forbes Avenue  
Pittsburgh, PA 15213*

## Abstract

We have modified the X11 windowing system to use the native communication facilities of the Mach 3.0 microkernel. Our new implementation can rely on Mach's low-overhead IPC facility as a direct replacement for sockets, or it can use shared memory as a transport between X11 clients and the server. On conventional BSD Unix systems, X11 communication is done through sockets. Because a user-level process implements Unix functionality on top of Mach 3.0, a socket-based version of X11 performs substantially worse than when running on a monolithic Unix kernel. Using Mach IPC as the transport between X11 clients and the server, X11 performance is slightly better than that of a monolithic system in which sockets are implemented inside the kernel as opposed to within a user level process. Using Mach's shared memory facilities as the transport, we have measured performance improvements of over 40%.

## 1 Introduction

Mach is a microkernel-based operating system that provides complete 4.3 BSD Unix emulation through a user-level Unix server [Golub et al. 90]. This approach allows existing Unix applications to run unmodified on top of the Mach microkernel. In many cases, the

---

This research was sponsored in part by The Defense Advanced Research Projects Agency, Information Science and Technology Office, under the title "Research on Parallel Computing", ARPA Order No. 7330, issued by DARPA/CMO under Contract MDA972-90-C-0035, and by the Open Software Foundation (OSF). Bershad was partially supported by a National Science Foundation Presidential Young Investigator Award.

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of DARPA, OSF, the NSF, or the U.S. government.

system has comparable performance. In some cases, though, Unix applications run more slowly on top of an emulated Unix than they do on top of an "in-kernel" version of Unix. Applications that currently suffer most are those that use the Unix socket interface. For Mach 3.0, Unix sockets can incur a great deal of overhead since a socket call must perform an RPC through the kernel to the Unix server. Since X11 depends on the socket interface, X11 applications can run noticeably more slowly on top of Mach 3.0 than on a conventional Unix system.

We have restructured the protocol-dependent layers of X11 [Gettys et al. 90] to rely directly on the communication mechanisms provided by Mach 3.0, rather than those provided by the Unix server. This approach allows us to improve window-system performance because it removes the Unix server from X11 client/server communication.

Our implementation has followed two different paths. In one case, we have implemented the X11 transport protocols using Mach IPC [Draves 90]. This yields performance slightly better, or comparable to that of an in-kernel implementation because it eliminates the context-switching overhead incurred by the out-of-kernel Unix system. In the second case, we have used shared memory for communication between X11 clients and servers reducing the system's reliance on kernel communication primitives [Bershad et al. 91]. This approach yields substantial performance improvements.

## 1.1 Motivation

Mach is a microkernel designed to provide a base operating system on which other operating systems such as Unix can be built. Two versions of Mach will be discussed in this paper. The first, Mach 2.5, includes the Mach microkernel and the Unix emulation code in the kernel's address space. This combination is comparable in speed to other Unix implementations such as Ultrix and BSD 4.3, since all the Unix code is in the monolithic kernel. Mach 2.5 is compatible with standard Unix, but also provides the added functionality of Mach. In contrast, Mach 3.0 contains just the Mach microkernel. A separate program, the Unix emulator, runs as a user-level process. Mach offers two kinds of communications channels, message passing and shared memory. Message passing is provided using the Mach IPC interface. Shared memory is provided using Mach's VM interface [Rashid et al. 87].

Under Mach 3.0, communication through a socket goes from the user code through the Mach microkernel, then to the Unix emulator. Therefore, more overhead is incurred for Unix socket system calls using Mach 3.0 than when using an in-kernel implementation of Unix. For example, on a 33Mhz 80486 system, a small (64 bytes) round-trip message using sockets takes 709 microseconds on Mach 2.5, and 2319 microseconds on Mach 3.0. Under Mach 3.0, using Mach IPC rather than socket code, the time drops to about 150 microseconds. With shared memory, the time to send a message can be as low as the time to format the message in a shared buffer.

Our goal was to retarget X11 to use the less expensive communication facilities provided by the Mach 3.0 operating system. We experimented with two different implementations, one using kernel-based IPC and one using shared memory.

X11 is a client-server windowing system that runs under many different operating systems and on many different hardware platforms. It provides a server that controls the machine hardware, including displays and input devices such as keyboards and mice. Programs access the server through a library of client functions. This library allows clients to request actions to be taken by the X server, as well as to request notification of events such as keyboard and mouse events. We were willing to make changes to the server, since there are relatively few servers, but were unwilling to do so to clients. We wanted to be able to take existing clients and use them under our modified transport, without having to edit or rewrite them. Therefore, our client-side changes are restricted to the X11 library, and require only that X11 clients be relinked.

## 1.2 The rest of this paper

The rest of this paper describes our experiences with restructuring the X11 protocols to use the native communication facilities of Mach. In Section 2 we discuss the use of Mach IPC as a replacement for Unix sockets in the context of the X11 protocols. In Section 3 we describe the use of a user-level communication protocol based on shared memory. In Section 4 we discuss the performance of the two approaches. Finally in Section 5 we present our conclusions.

## 2 Using Mach IPC

In our initial implementation, we replaced socket calls in the original X11 system with calls to Mach's IPC facilities. Structurally, this approach is similar to the original socket-based implementation on a monolithic kernel in that it uses kernel routines to pass messages between address spaces.

Most client requests in X are asynchronous, and are buffered at both the client and server end. For example, when a client requests that the server draw a circle on the screen, the client library may buffer the request. On the server side, the processing of the request may be delayed as well. There are two main ways that the client can synchronize its outstanding requests with the server's. The first is `Xflush`, which instructs the client to flush its buffers of any unsent requests to the server. The `Xflush` function returns after sending the buffered requests to the server, so it does not guarantee that they have been processed by the server. Stronger guarantees are provided with `Xsync`. `Xsync`, like `Xflush`, flushes the client buffer, but then requests the server to acknowledge that it has finished all pending requests. The client blocks until the acknowledgement arrives. Most clients generally send several display drawing requests, and then block waiting for user input, which causes an `Xflush`, or they make several display requests, `Xsync` to ensure that the display looks correct, and then make more display requests.

## 2.1 The X11 server

We modified the server to use the Mach nameserver for establishing a client rendezvous. The server creates a Mach port for client connections, and makes it available to clients through the name server. The server listens on the initial port for incoming messages. Upon receipt of a connection message, it creates a pair of Mach ports, and sends them to the connecting client. One of these ports is monitored by the server for requests from the client, and the other is monitored by the client for responses and events from the server.

An X11 server must monitor the activity of a potentially large number of clients. In the original version of the server, the clients were represented by a set of file descriptors representing sockets. The socket-based server monitored the connections using the Unix `select` call. The X server's `select` call also monitors Unix file descriptors pertaining to keyboard and mouse I/O. These I/O channels are relatively low-bandwidth and are not latency-critical. Consequently, we left their management to the Unix server even in the IPC-based implementation.

Under Mach 3.0, our IPC-based server maintains and listens on a port set, with one port per client. Because the slower I/O channels are accessed with Unix file descriptors, we could not simply replace the original server's `select` call with a call to check the status of the port set. On the server side, we introduced a second thread to handle the blocking I/O Unix call. The primary thread performs blocking receives on the port set, while another monitors I/O activity on the Unix file descriptors. When the second thread learns of pending I/O, it alerts the primary thread through a "back door" port. The primary thread wakes up and handles the I/O.

## 2.2 X11 clients

On the client side, the X11 connection code was replaced by a nameserver lookup, followed by a message to the connection port of the server. Socket writes were replaced with Mach port sends, sending a variable length array instead of writing to the socket in a stream format. Socket reads were replaced with receives from Mach ports.

X11 allows access to the descriptor that is used to access the X11 server. With sockets, this is a file descriptor, and is made accessible so that single-threaded clients can multiplex their activity across several I/O channels. For example, `xterm` uses the descriptor to perform system calls such as `select` in a set with TTY descriptors. When using Mach IPC, however, there is no real Unix file descriptor through which client/server communication passes. The mixing of file descriptors, which are a strictly Unix mechanism, with Mach ports, which are a strictly Mach mechanism created some difficulties for the implementation. Unlike the server, where all blocking I/O could be controlled since it occurred within a single "program," client behavior is unconfined.

We solved the "mixed metaphor" problem by exporting a pseudo-descriptor from the X libraries. The pseudo-descriptor is simply a small integer that represents the Mach port on which server communication is occurring. We provided library stubs for system calls to watch for the pseudo-descriptor. For example, `select` was written as a client-side stub

to check if the pseudo-descriptor was in the argument set. If not, a Unix `select` call is performed. If the pseudo-descriptor is the only element in the set, a Mach system call is performed to find the number of pending messages, and a result based on this number is returned. If there is a mixture of the port and other file descriptors in the `select` call, we alternate between a `select` with a small timeout and the Mach call until either one returns a status that would be consistent with `select` terminating, or the time value specified by the caller had passed.

Our strategy of slow polling in the client for a multi-way `select` is less than optimal, and is asymmetric with respect to the server. But, given our constraints of not modifying clients, we had few other options. One option, for example, would have been to spawn off another thread in the client to handle the blocking Mach call, and to write to a special notifier descriptor that was being monitored in all `select` calls (as is done on the server side). We chose not to do this because it would have required building all X11 clients as multithreaded applications. Many X11 clients and libraries though assume that they are running in a single-threaded address space. This affects their use of Unix signal mechanisms. By rewriting the `select` call, we were able to pass a handle back to the clients that they could treat as a socket descriptor without impacting the single-threaded assumptions.

### 3 Mach Shared Memory

Our second approach uses Mach's shared memory facilities for communication between X11 clients and the server. While X11 is a network-extensible window system, it is commonly used for communicating between programs and servers running on the same machine. In such cases, shared memory, rather than kernel-level IPC, can be used as an extremely low-latency communication channel. Through the use of external pagers, Mach allows memory to be mapped into multiple address spaces at once. Once the pages have been mapped, no additional kernel interaction is necessary when accessing the memory.

We use shared memory only for communicating from the client to the server. Communication in the other direction is implemented as in the previous section. There are several reasons for this asymmetry. First, the majority of data is communicated from clients to servers, and not the other way around. Second, most messages from the server to a client result in the client becoming the next process to run, for example, the server's response to an `Xsync` request. On the other hand, most requests from the client to the server can be buffered, allowing the client to run ahead of the server, thereby eliminating context switches.

We modified the server so that during the connection phase it allocates memory for a shared buffer, and then returns that shared buffer to the connecting client. Although the server has access to the shared buffers of all clients, clients themselves do not have access to other clients' shared buffers. Since only the client writes new data into the buffer and only the server reads it, there are no race conditions and no need for explicit synchronization. The client moves a head pointer forward when making a request, and the server moves a tail pointer forward when processing a request.

With the socket-based and IPC-based implementations, each data transfer operation between the client and server, which went through the kernel, could also result in a context switch, allowing the server to perform the outstanding requests. In contrast, the shared-memory implementation avoids the kernel on data transfer, eliminating the possibility of an “accidental” context switch. For example, flushing no longer results in the server being the next process to run, as it generally does with kernel-based communication. All context switches must be forced by applications, which run until they are either descheduled due to quantum expiration, until they block waiting for input, or their shared buffer fills.

The server thread is normally blocked on a message receive for a port set that includes a port used by clients to alert the server that there are X11 requests outstanding. Clients only notify the server by way of this port if they have issued X11 requests that have not yet been processed, *and* the X server has not yet already been notified that there are outstanding requests.

The server shares with all clients a single page of memory that contains a bit indicating whether or not the server is suspended. The server sets the bit before it blocks on the port set. Clients, subsequent to posting a message to the server’s wakeup port, clear the bit. In this way, multiple clients may post requests to the server while requiring only one IPC message.<sup>1</sup> When the server wakes up, it scans the buffers that it shares with clients to find and service requests. When all requests have been satisfied, the server goes back to sleep.

## 4 Performance

We used several programs to benchmark the various X11 configurations. All benchmarks were run on a Gateway 80486 system running at 33 megahertz. The motherboard had 16 megabytes of 70ns ram, with 64 kilobytes of 25ns cache, and a Diamond Speedstar video board. All tests were run with no other users logged in, and no other processes running that were not part of the benchmarks.

### 4.1 Microbenchmark performance

Our first benchmark is *muncher*, which is a program that comes with the MIT distribution. Muncher repeatedly does xors to a 256x256 window, and flushes its buffers to the server after each complete 256x256 xor (through the *Xsync* command).

We modified the client and server so that we could tell how much time was spent in each during a run of the program. We first ran the program 2500 and 10000 times. This allowed us to subtract off program startup and cleanup overhead. We then modified the server so that it could process each request either once or twice to determine the screen and server

---

<sup>1</sup>We are aware that the fact that clients can write the bit creates a potential denial-of-service situation (a client erroneously clearing the bit without sending a wakeup message would keep other clients from notifying the server). This has not yet become a problem, but if it does, we will implement one of several obvious solutions, including having the server, rather than the client, write the bit, or having the server periodically wakeup and check the input queues regardless of message traffic.



	2.5 sockets	3.0 sockets	3.0 ports	3.0 shared memory
Client time	1.1	1.1	1.1	1.1
IPC time	1.6	5.0	1.3	0.3
Server time	1.7	1.7	1.7	1.7
Screen time	0.3	0.3	0.3	0.3
Total time	4.7	8.1	4.4	3.4

Table 1: *Muncher performance (ms/iteration)*

overhead. We modified the library to send each request to the server either once or twice to determine IPC overhead. Lastly, we modified the server to write to a fixed single page in memory all requests that would otherwise have gone to video memory. This allowed us to approximate how much of the server time was spent in the server code, and how much was spent controlling the video board.

Table 1 summarizes the results of the measurements using four different implementations of the communication system: Mach 2.5 with sockets, Mach 3.0 using sockets, Mach 3.0 using Mach IPC, and Mach 3.0 using shared memory. The component and total times are for one full 256x256 xor. Taking the Mach 2.5 with sockets implementation as a baseline, 3.0 sockets take almost 75% more time to execute, while 3.0 ports offer about a 6% improvement, and the shared memory implementation offers a 25% improvement.

Any further speedup would have to come from a major rewrite of the client or server internals, or from faster hardware. The client and server each take a fixed amount of time regardless of the transfer protocol used. The data transfer time using shared memory is less than the time for this particular system to actually modify video memory. Any further gains beyond this shared memory version would be negligible, since removing the data transfer completely would only yield about an 8% gain over the shared memory version.

## 4.2 Macrobenchmark performance

We measured two macrobenchmarks to evaluate the impact that our changes had on application behavior. One macrobenchmark measured the time to send a large file (15000+ lines) to an *xterm* that had jump scrolling activated. Using sockets, this test took 1 minute and 19 seconds to complete under Mach 2.5, and 2 minutes and 15 seconds under Mach 3.0. Using Mach 3.0 ports, the test took only 1 minute and 17 seconds to complete, slightly better than the baseline and much better than the Mach 3.0 socket-based implementation. The Mach 3.0 shared memory version took 1 minute and 5 seconds (85% of baseline).

For applications that do not explicitly synchronize with the server, there can be an even greater improvement in performance. For example, the *maze* program distributed in the MIT release draws and solves a random maze as fast as it can. It only synchronizes with the server (for display update) when it has generated a solution. Using sockets, a draw-solve cycle took an average of 1.91 seconds with Mach 2.5, and 3.97 seconds under Mach

3.0. Under Mach 3.0 with ports, the time drops to 1.88 seconds, for an improvement of only about 2%. However, when run under shared memory, it takes only 1.10 seconds per iteration, for a savings of about 42%.

## 5 Conclusions

Mach's Unix server can successfully emulate Unix as a user-level application. In some cases, performance may not be as good as with a monolithic Unix system. In these cases, the Mach primitives can be used directly to attack performance bottlenecks. The work described in this paper demonstrates that ports and shared memory can be used by applications on Mach 3.0 to give performance comparable to or better than Mach 2.5.

## References

- [Bershad et al. 91] Bershad, B., Anderson, T., Lazowska, E., and Levy, H. User-Level Inter-process Communication for Shared-Memory Multiprocessors. *ACM Transactions on Computer Systems*, 9(2), May 1991.
- [Draves 90] Draves, R. P. A Revised IPC Interface. In *Proceedings of the First Mach USENIX Workshop*, pages 101–121, October 1990.
- [Gettys et al. 90] Gettys, J., Karlton, P., and McGregor, S. The X Window System, version 11. *Software – Practice and Experience*, 20(S2):35–67, October 1990.
- [Golub et al. 90] Golub, D., Dean, R., Forin, A., and Rashid, R. Unix as an Application Program. In *Proceedings of the Summer 1990 USENIX Conference*, pages 87–95, June 1990.
- [Rashid et al. 87] Rashid, R., Tevanian, Jr., A., Young, M., Golub, D., Baron, R., Black, D., Bolosky, W., and Chew, J. Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures. In *Proceedings of the 2nd Symposium on Architectural Support for Programming Languages and Operating Systems*, April 1987.

# Real-Time Mach Timers: Exporting Time to the User

*Stefan Savage and Hideyuki Tokuda*

*School of Computer Science  
Carnegie Mellon University  
Pittsburgh, Pennsylvania 15213  
{savage,hxt}@cs.cmu.edu*

## Abstract

The current CMU Mach 3.0 microkernel exports simple timestamp and delay abstractions through `host_get_time()` and a timeout parameter to `mach_msg()`. While this is sufficient for many purposes, it does not provide the precision or generality required for a variety of real-time applications. In this paper we describe extensions to CMU's Mach 3.0 which provide users with flexible time-based synchronization and timestamp services. Additionally, we will describe how timing and scheduling services are integrated to allow real-time applications to handle *timing faults*.

## 1 Introduction

Modern operating systems are expected to provide services to allow application programs to synchronize with the passage of time. The BSD UNIX interface includes `gettimeofday()`, `sleep()`, `signal()` and `select()` to provide this functionality[7]. Similarly, Mach 3.0 exports `host_get_time()` and a timeout parameter to `mach_msg()` [8]. While these timing services are sufficient for a broad range of general purpose applications, real-time applications frequently require greater precision and flexibility. Real-time applications may be loosely defined as applications in which the relationship between execution behavior and the passage of time directly impacts program correctness. Included in this class of software are applications for factory automation, military command and control, robotics, and the rapidly growing field of continuous media[6]. Timestamps, exact delays, periodic signals and timeouts are among the services which may be demanded by these types of applications.

There are several requirements for providing this kind of support. First, it is important to be able to measure time with a high degree of precision. Second, users must be able to synchronize with this high precision time source. Third and last, it is necessary to integrate scheduling and synchronization so that computation can be modified in the presence of *timing faults*, or errors in temporal correctness.

---

This research was supported in part by the U.S. NRaD under contract number N66001-87-C-0155, by the Office of Naval Research under contract number N00014-84-K-0734, by the Defense Advanced Research Projects Agency, ARPA Order No. 7330 under contract number MDA72-90-C-0035, and by the Federal Systems Division of IBM Corporation under University Agreement YA-278067. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing official policies, either expressed or implied, of NRaD, ONR, DARPA, IBM, or the U.S. Government.

We have developed this necessary functionality in Real-Time Mach, an extension to CMU's Mach 3.0 which provides predictable scheduling and priority consistent synchronization[15, 14]. Time services are exported to the user in the form of two abstractions: *clocks* and *timers*. Timers are active objects which allow users to synchronize with time in a variety of ways. Clocks are devices which measure the passage of time and support the use of timers to a particular degree of accuracy. In the next sections we will examine these two abstractions more closely and the motivation for their design.

## 2 Clocks

In first implementing our time services we were faced with two problems. The first was our desire to allow the use of additional time measurement hardware to be integrated seamlessly for those applications which required high precision. Second, we encountered a problem with the lack of resolution stability in Mach 3.0's representation of the host time. In order to support synchronized time in a distributed environment while still preserving monotonicity[12], Mach 3.0 provides the `host_adjust_time()` interface. This interface allows the host time to be skewed at resolutions under one scheduling clock tick. Consequently, the time between two scheduling clock ticks is not always measured as such. The solution we reached to these problems was to create the abstraction of clocks, which measure the passage of time. Clocks can be used for accurate timestamps, measurements of CPU usage, or for basing representations of the time of day. Timers can be bound to different clocks depending on application need which allows the simple introduction of high resolution timing hardware. Additionally the problem of the host time representation disappears since clocks represent a local, device specific time. "Host time" is a system personality specific notion and belongs outside the kernel along with operations on its skew.

### 2.1 Functionality

In essence, a clock is simply a piece of hardware which measures the passage of time. It was logical therefore to represent clocks as Mach devices. This limits the number of additional abstractions and interfaces in the kernel and facilitates the addition of specialized time measurement hardware.

Clocks are manipulated using the standard Mach 3.0 device interfaces. As with any device, a clock must first be opened using `device_open()` with the name of the clock device. The names for these devices are machine dependent so we have provided a universal macro, `CLOCK_REALTIME`, which maps to a baseline clock device on all platforms. Clocks are generally manipulated through the `device_get_status()`, `device_set_status()`, and `device_map()` interfaces although the particular functionality allowed is hardware dependent. Typical operations include getting the time, getting the resolution, setting the resolution, or mapping the clock into a user's address space.

### 2.2 Implementation

For all system architectures which we have supported to this point (i486 AT, DECstation, and SUN3) the baseline clock has been based on the scheduling clock tick. In the case of the i486 AT and the DECstation where the clock resolution may be altered, it was necessary to multiplex the kernel `clock_interrupt()` routine on top of this baseline clock device. For instance, when the baseline clock on the i486 AT is directed to increase its resolution to 1 ms, the scheduling clock tick frequency increases by 10. To preserve the illusion of the standard 10ms scheduling clock tick, the `clock_interrupt()` routine is executed only at every tenth interrupt. When the interrupt frequency is not an integral divisor of the scheduling clock tick frequency, the scheduling clock tick

will be temporarily skewed over the period of the modulo of the two. This seemed reasonable since the short term inaccuracy of this number has relatively minor effects, and quantum based scheduling is a poor choice for real-time programs anyway.

In general, a clock device contains an interrupt handler, a representation of the current time and a timer queue. Pending timers are inserted into this queue in time order. When a clock device receives an interrupt it checks the head of the queue to see if any timers have expired. If so it sets an AST to process the timers at a safe point. Additionally, clock devices may provide services to map representations of the current time into a user's address space, to change clock resolution, or to obtain higher resolution timestamps than are available via the mapped device interface.

There is a great variance in the timing hardware, especially with respect to achievable interrupt resolutions. The MC146818 clock chip on the DECstation, for example, will only interrupt at frequencies which are powers of 2. Alternatively, the I8254 clock chip on the i486 AT will interrupt at any multiple of 838ns up to .055 seconds. To retain machine independence in our interface yet preserve the power of the underlying hardware we let the user request clock resolution changes in terms of a desired resolution and an allowable skew from that resolution. If the clock driver cannot be set to a resolution within (desired\_resolution - skew) then it will return failure. The correct resolution may be queried later if this information is important to the user.

### 3 Timers

Clock devices are sufficient for high resolution timestamps but to provide blocking synchronization requires a stateful object. This prompted the design of a timer abstraction. Timers are kernel-exported objects with three properties, an expiration time, a synchronization action to be taken, and an activity state. An active timer will perform its synchronization action when its expiration time is reached.

A timer is created using the `timer_create()` call. It is bound to a task and to a clock device which will measure the passage of time for it. The task binding is useful for ownership and proper tear-down of timer resources. If a task is terminated, then the timer resources owned by it are terminated as well. It is also possible to explicitly terminate a timer using the `timer_terminate()` call.

Timers allow synchronization primarily via two interfaces. `timer_sleep()` is a synchronous call similar to the POSIX 1003.4 `nanosleep()`[13]. The caller specifies a time to wakeup and indicates whether that time is relative or absolute. Relative times are less useful for real-time software since program-wide accuracy may be skewed by preemption[4]. For example, a thread which samples data and then sleeps for five seconds may be preempted between sampling and sleeping, causing a steadily increasing skew from the correct time base. Timers which are terminated or canceled while sleeping return an error to the user.

`timer_arm()` provides an asynchronous interface to timers. Through it the user specifies an expiration time, an optional period, and a port which will receive expiration notification messages. When the expiration time is reached, the kernel sends an asynchronous message containing the current time to this port. If the timer is specified as periodic then it will atomically re-arm itself with the specified period relative to the last expiration time.

Lastly, `timer_cancel()` allows the user to cancel the expiration of a pending timer. For periodic timers, the user has the option of canceling only the current expiration, or all forthcoming expiration.

These last facilities, periodic timers and partial cancellation, are important because they allow an efficient and correct implementation of periodic computation. This permits a user level

implementation of Real-Time Mach's periodic threads.

## 4 Timing Faults

The original motivation for this work came from a limitation in Real-Time Mach's thread package. While users could specify a deadline for periodic threads, this deadline would only be checked at the end of a thread's periodic computation. Effectively this meant that the user was only notified at the start of the next periodic computation. Moreover, aperiodic deadlines were largely ignored. It was clear that we needed some sort of asynchronous notification mechanism to allow such timing faults to be acted upon aggressively. We looked first into the use of the exception mechanism, but its current design left open the possibility of extended priority inversion in the kernel. This is because the exception mechanism *knows* that the exception was caused by the current thread. Therefore a timing fault registered by a low priority thread could execute in the context of a high priority thread, and keep it blocked in the kernel indefinitely. Instead of radically changing the exception code we designed a new asynchronous mechanism which led to the creation of a generalized timer service.

The timer interface supports timing faults via a flag in `timer_arm()` which indicates that the calling thread should be suspended when timer expiration occurs. The interface is important because it allows timing fault handling to export exception-like semantics. When a timing fault occurs, the offending thread is suspended and a message is sent to its deadline handler which will then take corrective action and possibly resume the faulting thread. This action is left to the user, although in practice the most frequent actions include a change of scheduling parameters, user notification, or for hard real-time programs, thread termination. To simplify programming we have added library support for deadline handling in the context of our *rt.thread* package[15]. The following library routine:

```
thread_deadline_handler(mythread, mythread_attr, entry, arg)
```

is sufficient to insure that when *mythread* misses a deadline the handler (*\*entry*) (*arg*) will be invoked.

## 5 Performance

All benchmarks were performed on a Gateway 2000 486DX2 66Mhz system with 16 megabytes of 70ns ram, and 64 kilobytes of 25ns secondary cache. Both primary and secondary caches were warm. The system was running Real-Time Mach version MK78 with CMU Unix server version UX39 running in single user mode. Additionally, the network was disconnected and the benchmarks were scheduled with the highest thread priority. We used a STAT! [1] timer board to take measurements accurate to 1 $\mu$ sec.

### 5.1 Timer operations

We measured a series of microbenchmarks to evaluate the execution cost of each new timer operation. These measurements were repeated 1000 times and the average taken (controlling for clock and watchdog interrupts and their effect on the cache, the variance was very small)

Table 1 summarizes the latency of the timer operations. The measurements of `timer_arm()` are best case numbers in which there are no pending timers earlier in the clock queue. When other timers intervene, the cost increases linearly by a factor of .600ns per timer. The periodic form of `timer_arm()` is slightly longer than the one-shot but there is significant amortized savings since

operation	time in $\mu$ secs
timer_arm() [one-shot]	33
timer_arm() [periodic]	37
timer_cancel()	19
timer_create()	220
timer_terminate()	119
mach_thread_self()	20
null trap	7

Table 1: *Latency of timer operations*

expiration action	time in $\mu$ secs
unblock thread	106
send message	162

Table 2: *Latency of thread execution from time of interrupt*

timer reset costs are  $1\mu$ sec (again subject to the worst case  $O(n)$  insertion time for the clock queue). These numbers could be improved using tree-based or timing wheel algorithms[16].

The timer\_create() and timer\_terminate() are substantially slower than other primitives largely because they are involved in allocating memory and port structures. Additionally both of these operations are implemented using IPC while the rest of the timer operations are invoked directly through traps. The times for the mach\_thread\_self() and null traps are given for reference.

## 5.2 Execution latency

In table 2 we see the latency between the time a clock device interrupt occurs and a thread waiting on an expired timer starts to execute. The two cases represent two different timer expiration actions: threads which are directly blocked on a timer using timer\_sleep(), and threads which are waiting for a timer expiration message. MIG stub, copyout and general IPC overhead all contribute to the  $56\mu$ sec difference. We expect that a handcoded IPC stub in the kernel and general IPC improvements should make these numbers comparable. Also, we believe that the scheduling overhead component of these numbers can be reduced by at least  $40\mu$ secs.

## 5.3 Clock operations

The latency of clock device operations is bounded by the performance of the device server in the kernel. Table 3 shows a value of  $85\mu$ secs for timestamps via IPC, and this is typical for all clock

operation	time in $\mu$ secs
timestamp (using device_get_status)	85
timestamp (using mapped page/io port)	<2

Table 3: *Latency of clock timestamp operations*

operations. However, for some clock devices, changes in resolution may be postponed from the time of invocation for the duration of one clock device tick.

These numbers suggest that all timestamps should be obtained via mapped I/O, but this is not necessary true. For some clock devices the interrupt resolution differs by several orders of magnitude from the resolution observable from supervisor state. For example, i486 AT systems and some SPARC systems are able to obtain timestamps of accurate to 1-4 $\mu$ secs in the kernel, but periodic interrupts only occur every 1-10ms. In these cases, application programmers may wish to pay the extra overhead of invoking the kernel in exchange for increased precision.

## 5.4 Overhead

Clock device queue processing costs 1 $\mu$ sec at interrupt level. Changing the clock granularity increases the overhead of this routine proportionally. It has an adverse effect (sometimes disastrous) effect on locality of reference in the cache. For reference, the regular `clock_interrupt()` routine costs 9 $\mu$ secs before the addition of this code.

## 6 Related Work

POSIX 1003.4 defines both timer and clock interfaces which provided much of the inspiration for our own time interfaces in Real-Time Mach. Real-Time Mach's clocks and timers provide a superset of POSIX functionality, including the ability to change clock granularity dynamically and to effect scheduling decisions via the timing fault mechanism. Otherwise the interfaces are very similar.

Microsoft's NT executive provides *timer objects* for the purposes of time based synchronization [5, 11]. In addition to blocking time synchronization, timer objects may be associated with APC's (Asynchronous Procedure Calls) which allow the kernel to asynchronously set a thread's context to execute a particular function when a timer expires. Within the kernel and device drivers, timer objects may also be associated with DPC's (Deferred Procedure Calls) which perform a similar function but are invoked in the current execution context. NT timer objects invoke expiration actions using an AST-like mechanism much like Real-Time Mach does. Unlike our system however, NT does not provide the user with the ability to bind these objects to different clock devices.

The OSF RI has designed and implemented a similar set of interfaces for their microkernel called "clocks" [10, 9]. The OSF interfaces export services for getting timestamps, setting clock resolution, exact delays, and setting alarm messages. Alarms which are found to have expired at interrupt time are copied to a special alarm queue. A high priority kernel thread is unblocked and it sends alarm messages on behalf of the kernel. While there is substantial similarity between our work and that of OSF, we feel that the interfaces and mechanisms produced at CMU are smaller, more consistent with Mach 3.0, and have superior functionality. Briefly, CMU timers support periodic reset, time faults, and timer cancellation functionality which is not present in the OSF interface. CMU clocks are standard Mach 3.0 devices while OSF names their clocks via "clock\_id's" which are raw integers interpreted by the kernel. Lastly, CMU's implementation uses an AST instead of a kernel thread to dispatch timer messages, thereby avoiding the cost of additional queueing and context switch operations.

## 7 Current Status and Future Work

The previously mentioned time services have been integrated successfully into Real-Time Mach across a range of hardware. Specifically, we have supported the MC146818 clock chip on the



DECstation, the I8254 and MC146818 clock chips as well as the STAT! timer board for the i486 AT, and the standard SUN3 clock chip. The vast majority of this code is machine independent and adding new clock devices extremely simple.

We are working closely with the CMU Mach project to integrate this code along with the base Real-Time Mach code into the mainline CMU Mach 3.0 distribution. As a part of this merge some of the interfaces and mechanisms may change. With respect to this paper, we are investigating ways to allow timing faults to be handled within the exception mechanisms. Additionally, we hope that all kernel timing, such as device timeouts and pc sampling, can be subsumed by this single mechanism. In addition to removing duplication of effort this will simplify machine independent support for particular timing functionality, such as BSD's relatively prime profiling timer[7]. Lastly, we are examining ways in which the timer mechanisms can be integrated with first class user level threads[6, 2, 3] to provide low overhead real-time threads.

## 8 Conclusion

We have shown that there are three requirements for providing time services for real-time programs. These are, high accuracy time measurement, synchronization to these time sources, and integration with scheduling. We have implemented solutions to all of these problems in an interface which is clean, efficient and consistent with the Mach 3.0 design philosophy.

## References

- [1] Alpha Logic Technologies. *STAT! System Timing Analysis Tool User Guide*, 1992.
- [2] T.E. Anderson, B. Bershad, E.D. Lazowska, and H.M. Levy. Scheduler Activations: Effective Kernel Support for the User-Level Mangement of Parallelism. In *Proceedings of the Thirteenth ACM Symposium on Operating System Principles*, October 1991.
- [3] P. Barton-Davis, D. McNamee, R. Vaswani, and E.D. Lazowska. Adding Scheduler Activations to Mach 3.0. Technical Report 92-08-03, University of Washington, October 1992.
- [4] V. Blazquez, L. Redono, and J.L. Freniche. Experiences with "Delay Until" for Avionics Computers. *Ada Letters*, 12(1):65-72, January 1992.
- [5] H. Custer. *Inside Windows NT*. Microsoft Press, 1992.
- [6] R. Govindan and D.P. Anderson. Scheduling and IPC Mechanisms for Continuous Media. In *Proceedings of the Thirteenth ACM Symposium on Operating System Principles*, October 1991.
- [7] S. Leffler, M. McKusick, M. Karels, and J. Quarterman. *The Design and Implementation of the 4.3BSD UNIX Operating System*. Addison-Wesley, 1989.
- [8] K. Loeper. *Mach 3 Kernel Interfaces*. Open Software Foundation and Carnegie-Mellon University, January 1992.
- [9] K. Loeper. *OSF Mach Draft Kernel Interfaces*. Open Software Foundation and Carnegie-Mellon University, October 1992.
- [10] K. Loeper. *OSF Mach Draft Kernel Principles*. Open Software Foundation and Carnegie-Mellon University, October 1992.

- [11] Microsoft Corporation. *Preliminary Windows NT Device Driver Kit*, 1992.
- [12] D. Mills. Experiments in Network Clock Synchronization. DARPA Network Working Group Report RFC-957, August 1985.
- [13] Realtime Extension for Portable Operating Systems. Proposed Standard IEEE P1003.4/D12, February 1992.
- [14] H. Tokuda and T. Nakajima. Evaluation of Real-Time Synchronization in Real-Time Mach. In *Proceedings of the USENIX Mach Symposium*, November 1991.
- [15] H. Tokuda, T. Nakajima, and P. Rao. Real-Time Mach: Towards a Predictable Real-Time System. In *Proceedings of the USENIX Mach Workshop*, October 1990.
- [16] G. Varghese and T. Lauck. Hashed and Hierarchical timing Wheels: Data Structures for the Efficient Implementation of a Timer Facility. In *Proceedings of the Eleventh ACM Symposium on Operating System Principles*, November 1987.

# Adding Scheduler Activations to Mach 3.0 \*

Paul Barton-Davis, Dylan McNamee, Raj Vaswani, and Edward D. Lazowska  
Department of Computer Science and Engineering  
University of Washington  
Seattle, Washington 98195

## Abstract

When user-level threads are built on top of traditional kernel threads, they can exhibit poor performance or even incorrect behavior in the face of blocking kernel operations such as I/O, page faults, and processor preemption. This problem can be solved by building user-level threads on top of a new kernel entity, the *scheduler activation*. The goal of the effort described in this paper was to implement scheduler activations in the Mach 3.0 operating system. We describe the design decisions made, the kernel modifications required, and our additions to the CThreads thread library to take advantage of the new kernel structure. We also isolate the performance costs incurred due to scheduler activations support, and empirically demonstrate that these costs are outweighed by the benefits of this approach.

## 1 Introduction

User-level threads built on top of traditional kernel threads offer excellent performance for common operations such as creation, termination, and synchronization. Unfortunately, though, they exhibit poor performance or even incorrect behavior in the face of blocking kernel operations such as I/O, page faults, and processor preemption. This has presented the application programmer with a dilemma: use either kernel threads, which are well integrated with system services but have expensive common-case operations, or user-level threads, which have efficient common-case operations but are poorly integrated with system services.

To resolve this dilemma, Anderson et al. [ABLL92] designed a new kernel entity, the *scheduler activation*, which provides proper support for user-level thread management. User-level threads built on top of scheduler activations combine the functionality of kernel threads with the performance and flexibility of user-level threads. In the past few years a consensus has emerged that scheduler activations are the right kernel mechanism for supporting the user-level management of parallelism.

The goal of the effort described in this paper was to implement scheduler activations in the Mach 3.0 operating system [RBF<sup>+</sup>89]. This paper places emphasis on the implementation rather than the concepts involved. We describe the design decisions made, the kernel modifications required, and our additions to the CThreads thread library to take advantage of the new kernel structure.

### 1.1 User-Level Threads

When expressing parallelism, the less expensive the unit of parallelism, the finer the grain of parallel work that can be supported with tolerable overhead. It has long been accepted that kernel processes (e.g., as in Unix) are too expensive to support any but the most coarse grain parallel applications. Multiprocessor operating systems such as Mach and Topaz [TSS88] attempted to address this problem by providing kernel threads. But kernel threads have also proven to be too expensive, and Mach and Topaz now provide user-level thread systems built on top of their kernel threads. Anderson et al. [ABLL92] argue that the cost of kernel threads relative to user-level threads is not an artifact of existing implementations, but rather is inherent, arising from two factors:

---

\*This work was supported in part by the National Science Foundation under Grants No. CCR-9200832, CDA-9123308, CCR-8907666, CCR-8703049, and CCR-8619663, by Digital Equipment Corporation, and by the Washington Technology Center. The authors' email addresses are {pauld, dylan, raj, lazowska}@cs.washington.edu

- *The cost of kernel services.* Trapping into the kernel is more expensive than a simple procedure call. Architectural trends are making kernel traps relatively more expensive [ALBL91]. In addition, to protect itself from misbehaving user-level programs, the kernel must check each argument for bad values that could cause the system to crash. A user-level thread package uses procedure calls to provide thread management operations, avoiding the overhead of kernel traps. Furthermore, the package doesn't necessarily have to bulletproof itself since a crash will only affect the misbehaving program.
- *The cost of generality.* Kernel threads must be all things to all people. Even if a particular program has no need for some specific facility (e.g., round-robin scheduling), it still must pay the price for the existence of that facility. A user-level thread package can be customized and optimized for each application's needs [BLL88].

User-level threads have proven useful for expressing relatively fine-grain parallelism. Unfortunately, implementing user-level threads on top of existing operating system mechanisms (such as kernel threads) causes difficulties. The first problem is that the kernel threads are scheduled obliviously with respect to user-level activity; the kernel scheduler and the user-level scheduler can interfere with each other. The second problem is that blocking kernel events such as I/O, page faults, and processor preemption are invisible to the user-level. Various mechanisms have been suggested to address specific instances of these problems [MSLM91] [TG89] [Her91], but none of this work addresses all of the difficulties. Scheduler activations present a unified solution.

## 1.2 Scheduler Activations

Scheduler activations are an alternative to kernel threads for supporting the user-level management of parallelism. As presented in [ABLL92], a scheduler activations environment has the following key characteristics:

- *Processors are allocated to jobs by the kernel.* Processor allocation is managed by the kernel, based on information communicated from user-level. The kernel allocates a processor by providing the job with a *scheduler activation*, an entity much like a traditional kernel thread, but with additional properties noted below.
- *A user-level thread scheduler controls which threads run on a job's allocated processors.* Kernel-provided scheduler activations are simply vessels upon which the user-level scheduler multiplexes threads. Common operations such as thread scheduling and synchronization are performed efficiently at user-level, without kernel intervention.
- *The user-level notifies the kernel of changing demand for processors.* The kernel is notified when the job's parallelism crosses above or below its current processor allocation.
- *The kernel notifies the user-level scheduler of system events that affect the job.* These events include the allocation or preemption of a processor, and the blocking or awakening of an activation in the kernel. The kernel's role changes from *handling events* to being responsible for *communicating events* to the appropriate job's user-level thread management system. This allows the thread system to respond to the event in a manner most appropriate for the job.
- *The kernel never time-slices scheduler activations.* Scheduler activations are the means by which the kernel provides processors to jobs. A job always has exactly as many scheduler activations as it has physical processors. The kernel never multiplexes (a given number of) scheduler activations on (a smaller number of) physical processors.
- *Application programs can remain unmodified, yet take advantage of this new environment.* This is because event management is encapsulated within the user-level thread system, the interface to which remains unchanged.

### 1.3 Scheduler Activations in Mach

Anderson's prototype implementation involved modifying the Topaz operating system on the DEC SRC Firefly multiprocessor workstation [TSS88]. This provided an excellent prototyping environment, but the implementation was inaccessible to others, and experience was limited to systems with at most six processors.

Our goal was to integrate scheduler activations into a kernel that was widely used and that ran on a variety of platforms. We chose Mach 3.0, and conducted our work on a 20-processor Sequent Symmetry [LT88]. We envision that our implementation will fulfill several roles. First, it will allow final validation of the scheduler activations concept with a reasonable number of processors. Second, it will provide a base on which to conduct further investigations into processor allocation algorithms, thread scheduling algorithms, the integration of user-level threads with efficient communication primitives, the provision of services such as I/O and virtual memory management at user-level, and so forth. Third, it will serve as a model for those who wish to integrate scheduler activations with Mach on other platforms.

In undertaking this implementation, we made choices that favored expediency, Mach compatibility, and machine-independence. Toward the goal of expediency, we wanted to minimize modifications to the system, using existing kernel mechanisms whenever possible. To make the resulting system widely usable, we strove for backward compatibility with Mach; for example, we continue to support the existing kernel threads interface. Finally, we made the implementation as machine-independent as possible, so that it could easily be ported to other platforms. We sought to create a testbed for experimentation, rather than an entirely restructured system based upon scheduler activations. Maximal efficiency was not a goal of our implementation. Rather, we expect that experience with a production system will identify areas where optimizations will be most beneficial. We hoped that this approach would make our implementation more comprehensible and modifiable, as well.

### 1.4 Design Framework

In the scheduler activations model, each job's user-level thread system has control over scheduling on the processors allocated to it, and is notified if this allocation changes. The kernel's responsibility is to provide a scheduler activation (execution context) per processor allocated to the job, and to notify the job of any relevant events.

We implemented scheduler activations by modifying the behavior of Mach kernel threads. One key design decision that affects the structure of our implementation is the introduction of a processor allocation module, which in some sense replaces Mach's existing kernel thread scheduler. Mach's kernel thread scheduling policy is thread-based and quantum-driven. This policy is adequate for traditional workloads, but can be outperformed by a policy that takes advantage of the information conveyed between scheduler activations applications and the kernel.

The primary requirement of such a policy is the ability to manage processor allocation on a per-task basis, something difficult to do under the basic Mach policy. However, Mach's *processor sets* [Bla90] provided us with a mechanism to circumvent the standard thread-based policy (which is inappropriate for our purposes), and to replace it with our own task-based one. Processor sets are kernel entities to which tasks, threads and processors are assigned; threads execute only on processors assigned to their corresponding processor set.

Our design gives each task its own processor set. A policy module monitors the tasks' varying processor demands, basing its allocation decisions on these demands and on its own calculations of inter-task priorities. This new policy module could be implemented either in the kernel or in a user-level server. We chose the latter because it afforded us the flexibility to easily experiment with various policies. The final policy we chose (described later) is encapsulated in a user-level "processor allocation server", which uses existing kernel mechanisms to enforce its decisions: processors are allocated to a task by assigning them to the task's associated processor set, and are preempted by removing them from that set. As this design implies, our system consists of the following components:

- A set of kernel modifications implementing basic mechanisms, such as event notification. These are discussed in the next section.
- A user-level thread package that takes advantage of the new kernel, described in Section 3.

- A user-level server that manages processor allocation between tasks that are using scheduler activations. The policy used by this server is explained in Section 4.3.

## 2 Kernel Support for Scheduler Activations

This section explains the modifications we made to the Mach 3.0 kernel<sup>†</sup> (MK78) to support scheduler activations. From the perspective of the kernel, a scheduler activation is an ordinary Mach kernel thread, with the additional property that events caused by (or affecting) its task are reflected up to the user-level.

### 2.1 Initialization

Our modified Mach kernel supports both traditional Mach tasks (those desiring Mach kernel threads) and tasks desiring to use scheduler activations. A task informs the kernel of its desire to use scheduler activations by executing the following new system calls:

- `task_register_upcalls(task, upcalls)`. Registers a set of entry points in user space: the procedure addresses of the user-level upcall handling routines.
- `task_recycle_stacks(task, count, stacks)`. Provides the kernel with *count* blocks of user-space memory. When the kernel performs an upcall, it consumes one of these blocks for the user-level execution stack of the scheduler activation performing the upcall. It is necessary for the task to manage these stacks because the kernel is unable to detect when the information on a particular stack is no longer useful to the task. The task must ensure that stacks are always available for the kernel's use during upcalls (see Section 3.4.2).
- `task_use_scheduler_activations(task, TRUE)`. Sets a bit in the kernel's data structure for the task, marking the task as using scheduler activations. If a task has this bit – the *using\_sa* bit – set, then any kernel threads created within that task will have a (related) bit set. This bit – the *is\_sa* bit – signifies that the kernel thread is in fact a scheduler activation. The kernel uses these bits to decide how to handle particular events, as described below.

As a side effect of this call, the calling kernel thread (assumed to be the only one extant in the task) is converted into a scheduler activation. On return from this call, the task may proceed with its normal computation under scheduler activations semantics.

We control the task's state using multiple system calls in the interest of flexibility. Although this method results in slightly higher overhead at initialization time, this one-time cost seemed an acceptable price for allowing the task finer control over its state (allowing it to change its registered upcall handlers at any time, for example).

### 2.2 Handling Kernel Events

In the scheduler activations model, certain kernel events dictate that a notification be sent to the task. Notifications are implemented as upcalls from the kernel to user-level, with arguments as shown below. There are four notifications that can be sent from the kernel to the user-level:

- `blocked(new_sa, event_sa, interrupted_sas)`: *new\_sa* is carrying the notification that *event\_sa* has blocked. *interrupted\_sas* refers to activations that may have been interrupted in order to deliver the notification;<sup>‡</sup> as described in Section 2.2.1, this argument is typically `NULL` in the case of `blocked` notifications.
- `unblocked(new_sa, event_sa, interrupted_sas)`: *new\_sa* is carrying the notification that *event\_sa* has unblocked. *interrupted\_sas* refers to any activations that may have been preempted to provide the notification.

<sup>†</sup>No modifications to the UX emulator or server were required.

<sup>‡</sup>This field is called *interrupted\_sas* (plural) because an activation may block in the kernel while preparing a notification. In this case, several activations may be interrupted before the notification can finally be delivered.

- `preempted(new_sa, event_sa, interrupted_sas)`: `new_sa` is carrying the notification that `event_sa` was preempted due to a processor reallocation. `interrupted_sas` refers to any activations that may have been interrupted to provide the notification.
- `processor_added(new_sa, event_sa, interrupted_sas)`: `new_sa` is carrying the notification that a new processor has been allocated to this task. In this notification, both `event_sa` and `interrupted_sas` are NULL (see Section 2.2.4).

The implementation of these four notifications is described in the next four subsections; we first provide some necessary background.

A crucial component of a scheduler activation's state is its *user-level continuation*, which is similar to a kernel continuation (as described in [DBRD91]), but specifies a kernel routine to be run instead of returning to the user-level. We set this when we wish to gain control of the activation's execution in order to force an upcall to user-level (see Section 2.2.1).

A scheduler activation also has two fields (`event_sa` and `interrupted_sas`) that point at other activations, allowing it to carry information about them to user-level during a notification. This information includes the identities of the activations, the processor on which they were running, and their user-level register state.

The kernel uses the `thread_select()` routine to choose which thread or scheduler activation to run next. We maintain a flag per processor that tells `thread_select()` whether a notification is needed. The values of this flag include `PROC_SA_BLOCKED` for blocked events, `PROC_SA_HANDLING_EVENT` during notification, and `PROC_ALLOCATED` for processor reallocation between processor sets. When `thread_select()` notices that a processor's event flag has one of these values, it arranges to send a notification.

The next four subsections describe the handling of specific events.

### 2.2.1 Blocking

When an activation blocks, its `event_sa` is set to point to the activation itself, and its user-level continuation is set to the kernel routine `sa_notify()` (these are used when it unblocks; see Section 2.2.2). The processor's event flag is set to `PROC_SA_BLOCKED`. This causes `thread_select()` to create a new scheduler activation to carry the notification (instead of choosing some other activation to run). The new activation's `event_sa` is set to point to the blocked activation. The new activation starts at `sa_notify()`, which fetches a user-level stack on which to place the arguments for the upcall: the current activation and the blocked activation.

The user-level stack is manually set up to look like a trap return context, with the program counter set to be the task's upcall entry point, `sa_blocked()`. Finally, `thread_exception_return()` is used to begin executing in user space.

### 2.2.2 Unblocking

An unblocked event is started by the kernel's normal procedure for making an activation runnable after a blocking event (such as a disk read). One of the currently running activations is interrupted in order to run the unblocked activation; the policy driving this decision is described in Section 2.3.3. The interrupted activation calls `thread_block()` to yield the processor. `thread_block()` notices that the activation is preempting itself to pick up another runnable activation in the same task<sup>†</sup>, and sets the processor's event flag to `PROC_SA_HANDLING_EVENT`. Next, `thread_select()` finds the runnable unblocked activation, and because `PROC_SA_HANDLING_EVENT` is set, adds the interrupted activation to the unblocked activation's `interrupted_sas` list.

The kernel switches to the unblocked activation to allow it to clean up its kernel state (for example to call `pmap_enter()` after a page fault I/O is complete). This activation attempts to return to user-level by calling `return_from_trap()`, which checks to see if the activation has a user-level continuation. Since it does (it was set when the activation blocked), it calls the kernel routine `sa_notify()` instead of returning to the user-level. This routine uses the unblocked activation itself to perform the notification. It pushes the arguments to the notification: the current (unblocked) activation, the `event_sa` (set when the activation blocked to point to itself) with its user state, and any `interrupted_sas` with their user state. The user-level continuation

<sup>†</sup>As opposed to an activation preempted because its processor has been reallocated to another task.

is cleared, and the stack is set up to arrange return to the user-level `sa_unblocked()` handler. As before, `thread_exception_return()` is used to return to user-level.

### 2.2.3 Processor Preemption

Preemption notifications occur only when processors are reallocated from one task to another (by the processor allocator). The task losing the processor receives a preempted notification, as described below. The task gaining the processor receives a processor\_added notification, which is described in the next subsection.

The kernel is told by the processor allocator (via `processor_assign()`) to reassign a processor. If a scheduler activation is preempted as a result of the reallocation, we send a notification as follows. We create a new scheduler activation and start it at `sa_notify()`, setting its `event_sa` to be the preempted activation. The new activation is placed in the run queue of the task losing the processor; it is dispatched to user-level as with an unblocked notification, except that it begins user-level execution at a different entry point, the `sa_preempted()` handler.

Asynchronously to this activity, the preempted processor's `event` flag is set to `PROC_ALLOCATED`, and the processor is moved from the old processor set to the new one. If the task in the new processor set is using scheduler activations, it is notified as described below.

Notice that if a task loses its last processor, the notification of this event will not occur until the task again receives processors. It may be reasonable to suppress notification (rather than simply defer it) when a task has lost all of its processors – the last running activation could be resumed at its point of interruption, making the last preemption transparent to the task. There are cases where this is inappropriate: for example, a task managing cache affinity at user-level may require knowledge as to exactly which (as opposed to merely how many) processors it holds. We therefore chose to create notifications in all cases, allowing the task's thread scheduler to make its own decisions when the task is reactivated. While this is in keeping with the scheduler activations philosophy, tasks should perhaps be able to control this feature; this would allow them to avoid the (albeit small) user-level scheduling overhead when they feel it is unnecessary.

### 2.2.4 Processor Assignment

There are two methods of notifying a task of the allocation of a new processor, corresponding to whether or not the task has any runnable activations when the processor arrives. If there are runnable activations (they would represent undelivered preempted or unblocked notifications) they are simply allowed to return to user-level – on the new processor – as described above. The user-level thread system knows it has gained a processor because the notification does not refer to an interrupted activation, which is only possible if a new processor is carrying the notification.

If there are no runnable activations in the task, the kernel must create a `processor_added` notification. `thread_select()` tries and fails to find a ready activation to run. Before running the idle thread (as it would normally do), it checks the processor's `event` flag, which was set to `PROC_ALLOCATED` when the processor assignment occurred. In this case, it creates a new scheduler activation to carry the `processor_added` notification. The new activation, running in `sa_notify()`, simply fetches an upcall stack, pushing a pointer to itself for the `new_sa` argument but pushing `NULL`'s for `event_sa` and `interrupted_sas`. The user-level entry point is set to be the `sa_processor_added()` handler. The notification is sent to user-level via `thread_exception_return()`.

## 2.3 Implementation Details

The preceding has described the general operation of kernel event management. We now discuss in more detail some of the issues involved in its implementation.

### 2.3.1 Passing Scheduler Activation State

As described above, the register state of scheduler activations is passed from the kernel to the user-level thread management system. In our current implementation, this state includes all registers saved by the normal trap handler when the activation made the user- to kernel-mode transition. However, Mach's 80386 trap handler is careful to save the (large) state of the processor's floating point unit (FPU) only if this unit was in use when



the trap occurred. This leads to an obvious optimization in our notification path: FPU state is copied/passed only if this state had originally been saved by the trap handler.

### 2.3.2 Recycling Scheduler Activations

A new scheduler activation is created whenever an event occurs. To make this process efficient, the kernel maintains in each task a pool of kernel threads that can be quickly converted to scheduler activations. A tunable number of threads is initially inserted into this pool when the task informs the kernel that it will be using scheduler activations. New kernel threads – to serve as scheduler activations – are created only when the pool is empty. The kernel “recycles” activations automatically. A scheduler activation may be recycled whenever the kernel is sure that it holds no state that will be needed subsequently. More precisely, when notifying about an event, the kernel determines whether or not the associated activations may be recycled, as follows:

- **blocked:** A blocked activation cannot be recycled because it will have to run in the kernel when it unblocks.
- **unblocked:** The unblocked activation is used to perform the notification. This means that unblocked activations may not be recycled. If an activation was interrupted to deliver the notification, its user state is copied out to user-level, and it never runs again, so it can be recycled. (This is true of all interrupted activations.)
- **preempted:** A preempted activation’s state is copied out to user-level, and it will never run again, so it may be recycled. As explained above, we can recycle any interrupted activations associated with the notification.
- **processor\_added:** There is no *event\_sa* for *processor\_added* events, and no *interrupted\_sas* because the notification corresponds to a new processor’s arrival; therefore, nothing is available for recycling in this case.

### 2.3.3 Posting Notifications

A scheduler activation is the abstraction of a physical processor, and as such, activations are never time-sliced on physical processors by the kernel. Therefore, although scheduler activations are strongly based on kernel threads, the complicated scheduling machinery already in place for threads (priority manipulation, and so on) is irrelevant with respect to activations. In fact, it would be erroneous to use this machinery, because quantum-driven preemption within a scheduler activations task is explicitly to be avoided.

We modified the kernel so that thread priorities are ignored within tasks (processor sets) using scheduler activations. Preemption occurs only when there is a runnable but not running scheduler activation in the task. The kernel ensures that this only occurs as a result of particular events:

- a previously blocked activation unblocks.
- a processor is reallocated from this task to another, and a notification is necessary (see handling of preemption, described above).

Standard mechanisms are used both to check for the preemption condition, and to actually do the preemption when appropriate: the normal periodic checks in the former case, and the Asynchronous Software Trap (AST) mechanism in the latter. This implies that scheduler activations determine autonomously whether they should preempt themselves; the policy outlined above dictates that they do this only if there is a notification (represented by a runnable scheduler activation) to “pick up”. This met our goal of modifying the existing system as little as possible, and kept the advantage that scheduling decisions are made in a distributed fashion (in contrast to a possible scheme where the kernel chooses a particular activation to preempt, sets an AST for it, and forces it to pick up the notification).

Normally, the kernel uses priority to choose which activation to preempt. Our modified kernel, when dealing with scheduler activations, keeps track of which scheduler activations are not doing useful user-level work, and thus are candidates for preemption. We use an extra bit in the kernel thread structure,

*willing\_to\_yield*, which indicates whether or not the associated scheduler activation is “willing to yield” the processor. Since the kernel cannot determine this without help from the task, this bit is controlled using a new system call, `thread_willing_to_yield()`. If called with `TRUE`, the calling activation’s *willing\_to\_yield* bit is set to `TRUE`. Also, a per-processor-set count (*willing\_count*) is incremented, indicating that one more scheduler activation is in this state. If called with `FALSE`, or if the “willing” activation blocks for any reason, the activation’s *willing\_to\_yield* bit is set to `FALSE`, and the processor set’s *willing\_count* is decremented.

This notion of “willing to yield” is used to determine which scheduler activation should pick up a pending notification. In unmodified Mach, a kernel thread sets an AST for itself if it finds a higher-priority thread in its processor set. In our scheme, scheduler activations ignore priorities, preempting themselves to pick up notifications in the following way:

- If the current number of runnable activations in the processor set exceeds the number of activations “willing to yield”, then the activation finding this to be true sets an AST for itself. (That is, that activation allows itself to be preempted.)
- Otherwise, the activation checks whether it is “willing to yield”. If so, it sets the AST, preempting itself. The *willing\_count* is not decremented until the preemption is complete; this ensures that other activations do not unnecessarily preempt themselves because the *willing\_count* appears too low (while the preemption is in progress). If *runnable\_count*  $\leq$  *willing\_count* but the current activation is not “willing to yield”, it does not preempt itself. Instead, it simply assumes that either a) preemptions to pick up the runnable activations are already in progress, or b) other “willing” activations will eventually<sup>†</sup> notice that a preemption is required, and initiate one themselves. If one (or more) of the willing activations becomes busy before noticing the need for preemption, then the next busy activation to check the preemption condition will find *runnable\_count*  $>$  *willing\_count*, and preempt itself.

The scheme described here introduces some latency in notification. Some of this latency is inherent in the existing system: noticing the need for preemption has some delay, and performing the preemption requires some time. We felt that better decisions could be made using the current distributed decision-making scheme than by having the kernel choose specific activations to preempt (in the latter case, the activation’s status could change after the preemption request was set but before the activation noticed it, resulting in the preemption of a usefully busy activation rather than another, potentially idle, activation). Extra latency is incurred when there are the same number of “willing to yield” activations as runnable activations, but some of them become busy before all of the notifications have been picked up. The extra latency in this uncommon case was deemed an acceptable price for ensuring the proper activation is preempted.

### 2.3.4 Kernel Threads vs. Scheduler Activations

Scheduler activations can masquerade as kernel threads by turning off their *is\_sa* flag. A scheduler activation thus disguised is allowed to block without causing a notification to be sent. This is done for brief periods in the kernel to prevent notifications from being posted at inopportune times. Consider what would happen if an activation page faults in the kernel during a `blocked` notification. If its *is\_sa* bit was on, this would be a new event, which would cause another `blocked` notification, which could cause another fault, and so on. We prevent this by turning the *is\_sa* bit off until (just before) the activation reaches user space.

We currently restrict tasks that are using scheduler activations to use them exclusively<sup>‡</sup>. This is not a necessity, but rather a decision made to simplify the implementation. The primary reason for this constraint is the difficulty of integrating scheduling policies for scheduler activations and kernel threads.

In order to allow suspension of scheduler activations tasks, we have had to make some of the task manipulation code aware of scheduler activations use. Without any special action, each activation would cause a notification as it suspended, and the notification would continue running. We wanted to support suspension to make scheduler activations tasks backward compatible with Mach tasks, and because task suspension is a normal part of task termination. The task manipulation code has been modified to toggle the `task->using_sa` flag appropriately. `task_hold()` turns off `task->using_sa` (saving the value of the

<sup>†</sup> The next time they check for ASTs.

<sup>‡</sup> One possible use for combining kernel threads with scheduler activations in the same task is as a debugging tool. For example, while testing upcall handlers, it may be useful to prevent them from generating additional upcalls.

bit) so we can suspend activations without notification. `task_release()` resets `task->using_sa` from the saved value. `task_terminate()` turns off the bit permanently.

### 3 User-Level Threads

CThreads [CD88] forms the basis of our user-level thread package. We chose CThreads because it already works with the Mach kernel and has been widely used in the Mach community. CThreads is implemented as a library of functions with which the application program is linked.

The interface exported by our modified CThreads library is the same as that of standard CThreads<sup>†</sup>. The following sections describe how we have changed the CThreads package to take advantage of the new kernel and present some details of the implementation.

#### 3.1 CThreads with Scheduler Activations

The “Mach Thread” implementation of CThreads [CD88] runs user-level threads on top of Mach kernel threads – when a program’s parallelism increases, it asks the kernel for more kernel threads, which are used to run user-level threads. Our CThreads implementation continues to multiplex user-level threads onto single kernel entities (scheduler activations instead of kernel threads), but changes the unit of resource requests from threads to processors. When we ask for a processor, what we actually receive is a new scheduler activation, executing our `sa_processor_added()` handler on the newly-allocated processor.. Thus, the kernel is responsible for providing scheduler activations to run on the processor, and the thread package assumes responsibility for deciding what they should do. In summary, our model involves:

- user-level threads
- multiplexing of user-level threads onto scheduler activations, using conventional techniques
- a number of scheduler activations, created by the kernel and guaranteed to be the same as the number of currently allocated processors

#### 3.2 User-Level Components

In order to adapt a user-level thread package to a scheduler activations kernel, a number of components must be added. These include a new initialization phase, new actions at thread creation time, and code to handle upcalls from the kernel. This section details the operation of these components.

At initialization time, our startup code hands to the kernel the addresses of the handlers for the four types of notifications. It then creates a set of user-level threads that will be used to execute these handlers (via the upcall mechanism described in Section 2.2), and marks them as being upcall-handling threads by setting their *is\_upcall* bit. It passes the addresses of these threads’ stacks to the kernel (via `task_recycle_stacks()`). Next, it contacts the processor allocation server, which creates a processor set for the task and allocates a processor on which the task can run. Finally, the current kernel thread tells the kernel that the task is using scheduler activations, becoming a scheduler activation as a result.

Whenever new user-level work is created, the thread system decides whether to ask for more processors. If the processor allocator gives the task another processor, the kernel ensures that the task will eventually receive a `processor_added` notification; the scheduler activation carrying the notification is then used to execute user-level work on the new processor.

When notified that a scheduler activation has blocked in the kernel, the thread system uses the notification to dequeue and run the next ready thread.

When notified that a scheduler activation has unblocked in the kernel, the thread system determines which (associated) user-level thread has unblocked, and marks it as such. It then attempts to deal with threads holding locks (as explained in subsequent sections). Next, any threads found to be runnable are enqueued in a ready queue; finally, the next ready thread is run.

<sup>†</sup>Except that `cthread_wire()` and `cthread_unwire()`, which bind and unbind user-level threads to particular kernel threads, do not have meaning when running with scheduler activations.

If a processor gets taken away from a task, that task receives a preemption notification on one of its remaining processors. When this happens, the preempted thread is handled according to the lock resolution policy, then put on the ready queue. Finally the next ready thread is dequeued and run.

### 3.3 Critical Sections

Critical sections pose problems for both kernel and user-level threads. The problem arises because preemption can occur while a thread is in a critical section, and this can impede the progress of other threads that wish to enter the critical section.

Various solutions to this problem have been proposed [ABLL92] [BRE92] [Her91]. Each of these solutions makes certain assumptions regarding the prevalence and characteristics of critical sections, and each requires a certain degree of system support, entailing a certain amount of implementation complexity. We chose to defer the evaluation of which (or which combination) of these was the best choice for our purposes, instead selecting a temporary solution that, while imperfect, was attractively easy to implement.

The basis for all user-level activity in our system is CThreads, which makes widespread use of spinlocks to guard critical sections. Our scheme therefore focuses on allowing the system to quickly restart threads that have been preempted while holding spinlocks. We rely upon the use of a per-thread lock counter: lock acquisition increments the counter, while lock release decrements it. Our upcall handlers check the lock counts of all the user-level threads referenced by their arguments, and attempt to run each runnable, lock-holding thread until its lock count reaches zero. This is done *before* attempting to acquire any locks, to prevent deadlocks caused by a preempted thread holding a run queue lock. We return control to the scheduler if we become the subject of a notification while spinning on the lock (we block or are preempted), when the last lock is released, or after failing to acquire a spinlock after a number of attempts.

Further details of our lock handling strategy are presented in Section 3.4.3.

### 3.4 Implementation Details

We used the “Mach Thread” implementation of CThreads as the basis of our thread library. This implementation is optimized for use on top of kernel threads, in that it takes measures to explicitly block kernel threads not busy with user-level work. However, a task’s need to create more kernel threads than it has processors is an artifact of the poor support for parallelism provided by traditional kernels. That is, the task must create additional kernel threads so that its processors can be kept usefully busy if one of its active kernel threads should happen to perform a blocking operation (or page fault). Further, the task must ensure that these “reserve” kernel threads remain blocked until needed, lest the kernel’s time-sharing policy cause them to interfere with their usefully busy colleagues.

A scheduler activations kernel eliminates the need for the task to concern itself with such kernel thread management issues. The kernel ensures that the task has exactly as many scheduler activations as it does physical processors. The task need not create “reserve activations” because the kernel further ensures that the task receives notifications of blocking events: when one of the task’s activations blocks, the kernel returns the processor to the task, along with a new activation that may be use to run other user-level work.

A scheduler activations kernel thus simplifies this aspect of the implementation of a user-level threads package. We were therefore able to streamline CThreads by eliminating its kernel thread management code and data structures (in particular the `cproc` structure, used by the original package to represent kernel threads).

Scheduling in the resulting package is simplified because we need deal with only one type of entity: a user-level thread. We do, however, differentiate between threads that are handling upcalls and threads that are performing other work. Upcall-handling threads are run in preference to those doing user-level work because the scheduler needs to maintain an accurate view of the system. Our implementation retains CThreads’ FIFO scheduling of threads running user-level work. Threads are run according to the following policy:

- if there is a thread in the stack of upcalls, pop and run it.
- otherwise, if there is a thread in the run queue, dequeue and run it.
- otherwise, spin for some time waiting for additional work, then mark this processor as “willing to yield”.

- spin, waiting for work to arrive, when it does, mark this processor as busy, and run the work.

The following sections describe other details of our implementation.

### 3.4.1 Stack-Based Identification

One of the first duties of each upcall handler is to identify the user-level threads affected by the event(s) that its arguments describe. We considered two schemes for this:

- the existing CThreads mechanism of using a pointer stored at the base of the user-level stack
- a mapping between scheduler activations and the user-level threads they are running

We decided to use the CThreads scheme, although it caused some implementation difficulties (see below), because we thought it easier than trying to maintain the consistency of a mapping between scheduler activations and user-level threads. Instead, we find the base of the stack (which is simple, because CThreads stacks are aligned on power-of-2 addresses), and then offset to find a pointer to the thread control block, thus identifying the thread.

Although the stack-based identification scheme appears simple, it is complicated by the existence of the Unix system call emulator [GDFR90]. This is a body of code mapped into the address space of each task created by the Unix server. When a task makes a Unix system call, a trap instruction carries it into kernel space. The kernel realizes that the system call is to be emulated, and returns to user space, but in the emulator. Here, the emulator code switches stacks before performing whatever operations are necessary to carry out the call (such as sending a message to the Unix server).

This means that if an activation blocks while executing emulator code, its user-level context could include a stack pointer that does *not* point at a properly initialized stack. If in the `sa_unblocked()` handler we try to determine our user-level thread identity using the current stack, we would at best get an incorrect answer, and more likely generate an exception. To deal with this, we have to be able to identify emulator stack pointers correctly, and understand how to determine the “real” user-level stack from them.

This introduces an unfortunate dependence between our CThreads package and the operation of the emulator. Modifying the emulator code to properly initialize the new stack would eliminate this problem. However, we were unsure whether such a change would have to be made in all emulators (i.e., emulators providing environments other than Unix) that run on top of Mach. Under the assumption that this were true, we chose to handle the problem in CThreads (which we had already modified) rather than to depend upon emulator alterations.

### 3.4.2 Stack Management

We mentioned in Section 2.1 that at startup, we provide the kernel with a set of stacks for use when running notifications. Since we preallocate a finite (perhaps even small) number of stacks, we need to ensure that the kernel does not run out as we receive successive notifications. We do this by passing stacks back to the kernel when they are no longer needed (using the `task_recycle_stacks()` system call). This is done from within the context switch function used to leave an upcall handler. Once we have switched to the new stack, we check to see if the old one belonged to an upcall handler; if so, we recycle it. Some caching of these stacks is done at user-level, so that we do not incur the overhead of a system call for every context switch out of an upcall handler.

### 3.4.3 Lock Handling Details

As described in Section 3.3, when a thread that holds a spinlock is preempted, the upcall handler that processes the notification is usually able to continue the lock-holding thread until it releases its last lock. When none of the threads in the notification hold any spinlocks, the upcall handler is able to acquire the global run queue lock<sup>†</sup>, and enqueue the runnable threads. At this point, the upcall handler is finished, so it exits, handing off to the next ready thread.

<sup>†</sup>It may have to spin for a while if the lock is held on another processor, but it is safe from deadlock.

Occasionally, an upcall handler that is processing preempted lock-holding threads may itself be preempted. This scenario is depicted in Figure 1. The situation depicted could have occurred due to a processor reallocation, as follows: Upcall handler  $U_1$  was handling the lock-holding threads  $T_1$  and  $T_2$  (holding locks  $L_A$  and  $L_B$  respectively) when it was preempted, because the processor it was running on was assigned to another task. Delivering the notification of this preemption event interrupted another lock-holding thread,  $T_3$ .  $T_3$  already holds lock  $L_C$ , and was preempted while trying to acquire  $L_B$ .

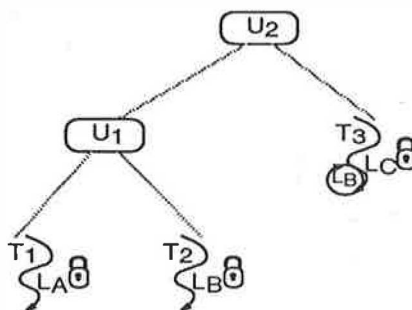


Figure 1: Nested Upcall Handling

In the general (though rare) case, the tree of upcall handlers is more than one level deep. The threads at each level of the tree may be waiting for locks held by threads at any other level of the tree. To prevent deadlock in this situation, it is sufficient to traverse the tree, in order to run the lock-holding threads until each thread is able to release all of its locks. The traversal is made possible by having the spinlock acquire code return control to the upcall handler continuing the thread after a number of failed spins<sup>†</sup>, so it can attempt to continue other lock-holding threads.

In the example shown in Figure 1, if  $U_2$  continues  $T_3$ ,  $T_3$  will spin unsuccessfully trying to acquire  $L_B$ , eventually returning control to  $U_2$ .  $U_2$  continues  $U_1$ , which in turn continues  $T_1$  and  $T_2$ .  $T_1$  and  $T_2$  will both release their locks after completing their critical sections.  $U_1$  then requeues  $T_1$  and  $T_2$  on the ready queue and exits, handing off back to  $U_2$ , which is now able to successfully continue  $T_3$ .

Multiple processors may be simultaneously busy running upcall handlers that are continuing lock-holding threads. Threads on one of the processors may be trying to acquire locks held by threads on another processor and vice versa. Even this will not result in deadlock, because each processor guarantees that all lock-holding threads are able to make progress. For this guarantee to be valid, each lock-holding thread must be either running on a processor or the subject of a notification, so an upcall handler is able to continue it. This condition holds because properly structured programs do not allow threads to block on user-level synchronization operations while holding a spinlock.

### 3.5 Possible Optimizations

As in the case of the kernel, the user-level implementation leaves many opportunities for optimization. In particular, although our package differs (internally) from standard CThreads, it might benefit from the improvements to the latter that have been recently suggested [Dea93]. As described in section 3.4, our package already includes one of these optimizations, that of having idle threads spin (rather than block) waiting for work. We are currently experimenting with the other ideas proposed in [Dea93].

The remainder of this subsection discusses the influence of a scheduler activations environment on other possible optimizations.

#### 3.5.1 Trap State Restores

The low overhead of context switching at user-level is one of the principal attractions of user-level thread systems. In the common case (a thread switching due to user-level synchronization or termination), our CThreads retains this low overhead. However, when context switching to a thread that was previously the

<sup>†</sup>Threads that aren't being continued by an upcall handler continue to spin until they acquire the lock.

subject of a notification, the overhead is increased somewhat. This is because the state that we must restore was saved by the kernel trap handler when the thread crossed into the kernel (i.e., the complete register state,<sup>†</sup> rather than just scratch registers). We are provided with the complete state as part of the upcall arguments, and use it without further modification during our context switch routine in a manner similar to the kernel's `thread_exception_return()`.

On the Sequent, some of the register restores are probably redundant, given Mach's use of the 80386's flat address mode [Int87]; in particular, it seems likely that the segment registers do not need to be restored by every user-level context switch, since they are constant across all activations in a task. Avoiding these redundant restores, as well as careful attention to register usage in the upcall handlers, could reduce the switch overhead.

### 3.5.2 Per-Processor Run Queues

Per-processor queues of ready threads are a commonly suggested optimization technique [ALL89] [FL89]. This could fairly easily be implemented on top of our system, but with mitigated benefits. In a non-preemptive threads package, per-processor ready queues can allow queue operations to proceed without synchronization, significantly improving performance. With scheduler activations, however, notifications can come at any time, possibly interrupting threads manipulating run queues; processing notifications therefore requires synchronized access to run queues. Even so, this may be a worthwhile optimization if the run queue is a bottleneck.

## 4 Performance

The goal of scheduler activations is to combine the functionality of kernel threads with the performance and flexibility advantages of managing parallelism at the user level within each task. To evaluate the effectiveness of our implementation, we first measure the costs associated with delivering notifications of kernel events. Next, we isolate the impact of preemption, demonstrating that scheduler activations allow applications to overcome its detrimental effects. Finally, we measure the benefits of scheduler activations in practice, using a multiprogrammed workload consisting of several concurrent copies of a real scientific program.

### 4.1 Upcall Performance

We measure the cost of notification as the time from the start of notification creation to the arrival of the notification at the user-level. The cost of notification can be broken into the following components:

- *kernel overhead*: the time required to return from kernel to user-level.
- *notification preparation*: the time to create and prepare a scheduler activation for notification, not including the data movement cost.
- *data movement*: the time to push the arguments, including activation state, onto the notification stack.

Table 1 shows a breakdown of the cost of each type of notification. Since each notification follows an identical path to transfer from kernel to user space, the kernel overhead is the same for each. Small differences in code paths account for the (correspondingly) small differences in notification preparation time. The major difference between the costs of the notifications – and the major contributor to the cost of each – is the data movement cost.

For `blocked` notifications, state is passed for the new activation (carrying the notification), but only minimal state is passed for the blocked activation (the `event_sa`). This is because this state is not useful to the user-level thread scheduler until the activation has unblocked; the purpose of notification in this case is not so much to inform the task of an interesting event, but rather to return the processor so that it may be used while the old activation is blocked. For this reason, `blocked` activations incur about the same data movement cost as `do_processor_added` notifications, which have no `event_sa`.

<sup>†</sup> As previously explained, "complete register state" does not necessarily include the floating-point registers, which are passed by the kernel only when appropriate (see Section 2.3.1).

Upcall type	total latency	kernel overhead	notification preparation	data movement
blocked	520	90	230	300
unblocked	915	90	225	600
preempted	860	90	170	600
processor_added	490	90	170	230

Table 1: Upcall Latencies (times in microseconds)

The unblocked and preempted notifications, however, are more expensive. Each such notification involves an *event.sa* whose state must be passed to user-level; furthermore, delivering either of these notifications typically requires interrupting a running activation, and the state of this activation must also be copied out to user space. The data movement cost associated with unblocked and preempted notifications is therefore much higher than that incurred by the two previously discussed.

In the remainder of this section, we demonstrate that the benefits of using scheduler activations outweigh the cost of posting (and handling) notifications.

## 4.2 The Cost of Preemption

When a processor is preempted from a task, the kernel can deal with the (kernel) thread running on that processor in three distinct ways:

- *The kernel thread is suspended for the duration of the preemption.* Since the kernel thread is executing some user-level thread, that thread also remains suspended until the task receives another processor.
- *The kernel thread competes for the remaining processors.* The kernel thread scheduling policy handles the mismatch between kernel threads and processors by multiplexing the threads on the processors (e.g., using time-slicing). User-level threads make progress whenever the kernel policy happens to dictate that their associated kernel thread is run.
- *Scheduler activations are used to notify the task of the preemption.* The task's thread scheduler is given the state of the associated user-level thread, which may now be run according to the user-level scheduling policy.

The first of these approaches (suspension) can lead to extremely poor performance if the suspended thread is critical to the progress of the application. We illustrate this by using an artificial benchmark: a simple “pingpong” program consisting of two user-level threads alternating access to a shared data structure. This task is initially assigned two processors, and runs in its own processor set to isolate it from other activity on the machine. We arranged processor preemption so that the task experiences a fixed number of preemptions (50), each lasting for  $M$  msec.

Figure 2 shows the elapsed time for this program under the three approaches (*suspension*, *time-slicing*, and *scheduler activations*) for various values of  $M$ . Included is a “reference” line denoting the time required to complete the job on two processors in the absence of preemptions.

Although the benchmark used here is an artificial one, it exemplifies an application that is sensitive to the delaying of any of its threads. The results for the *suspension* case in Figure 2 therefore reflect the “maximum” cost of a preemption, since in this case the application quickly becomes paralyzed if either thread is delayed. The *time-slicing* case performs much better, because the kernel policy ensures that the delayed thread does make progress. However, in this case, kernel threads are scheduled without regard to the user-level work they may be performing. Better performance is achieved by the *scheduler activations* case, in which the task's thread scheduler is allowed to extract the state of the delayed user-level thread, after which the thread scheduler can make its own (wiser) decision as to which thread should be allowed to execute.

These results suggest that for this artificial benchmark, the detriment of oblivious kernel scheduling, while noticeable, is relatively small. In the next subsection, we show that this is a much more important factor in practice.



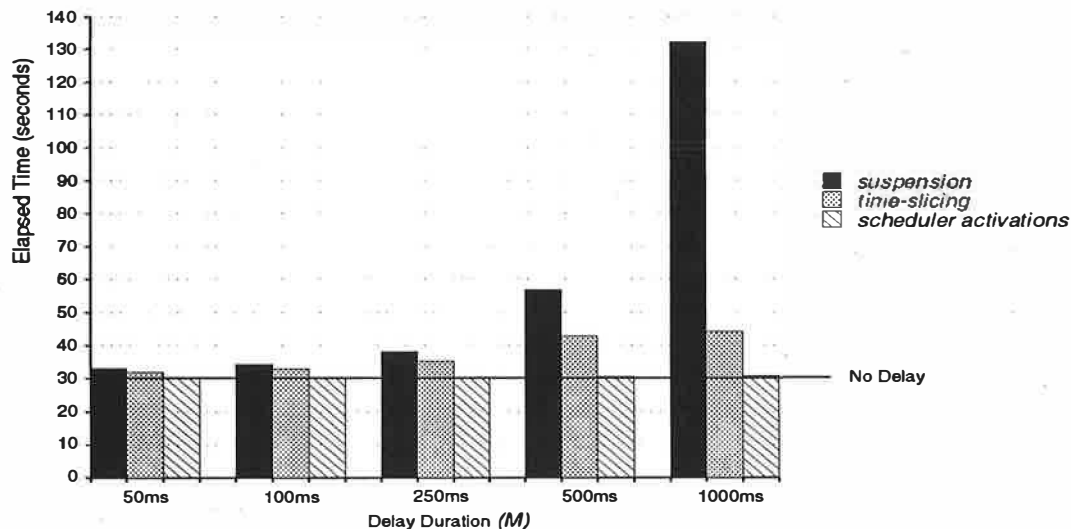


Figure 2: Effect of (50) Preemptions of Varying Length

### 4.3 Application Performance

To demonstrate the advantages of scheduler activations over oblivious kernel scheduling, we study the average elapsed time experienced by a real application running in a multiprogrammed environment.

The application we use, called *Gravity*, is an implementation of the Barnes and Hut clustering algorithm for simulating the gravitational interaction of a large number of stars over time [BH86]. The program contains five phases of execution; these phases are repeated for each timestep in the simulation. The first phase is sequential, while the last four are parallel. Between each of the parallel phases is a barrier synchronization at which the parallelism decreases briefly to one.

We derived the speedup curve for the application, finding that performance was best if it were allowed to run (in isolation) on 12 processors. We then arranged to run several (1-5) concurrent instances of *Gravity* under two different processor allocation policies. We first ran the jobs using the existing kernel thread scheduling policy: each job created a kernel thread per processor desired (12), and the jobs were simply allowed to run to completion on our (20-processor) machine.

Next, we used our processor allocator to provide a simple “space-sharing” policy, of the sort previously shown to provide good performance for this type of workload on this class of machine [TG89, MVZ90]. This allocator simply divides the available processors equally amongst the jobs: given  $P$  processors and  $J$  competing jobs, each job is allocated  $P/J$  processors. On each job arrival, the allocator computes a new equipartition allocation, and processors are preempted from active jobs for reassignment to the new arrival. Preemptions are coordinated by using scheduler activations to notify active jobs of processor loss. The jobs’ thread schedulers therefore have immediate access to interrupted threads, and the kernel ensures that the jobs always have exactly as many activations as processors (avoiding oblivious kernel scheduling).

Note that, as described in [MVZ90], better performance can be achieved by having the processor allocator respond to dynamic changes in the jobs’ parallelism (i.e., processor demand) by reallocating processors from jobs that cannot currently use them to jobs that can. Although we have implemented such a policy, we do not avail ourselves of it here. Further details regarding our user-level implementations of processor allocation policies can be found in [BMVL92].

Figure 3 shows the average runtime of each of the simultaneous runs of *Gravity* as the number of concurrent (competing) jobs increases from one to five.

These results demonstrate that user-level threads built on top of kernel threads can yield extremely poor

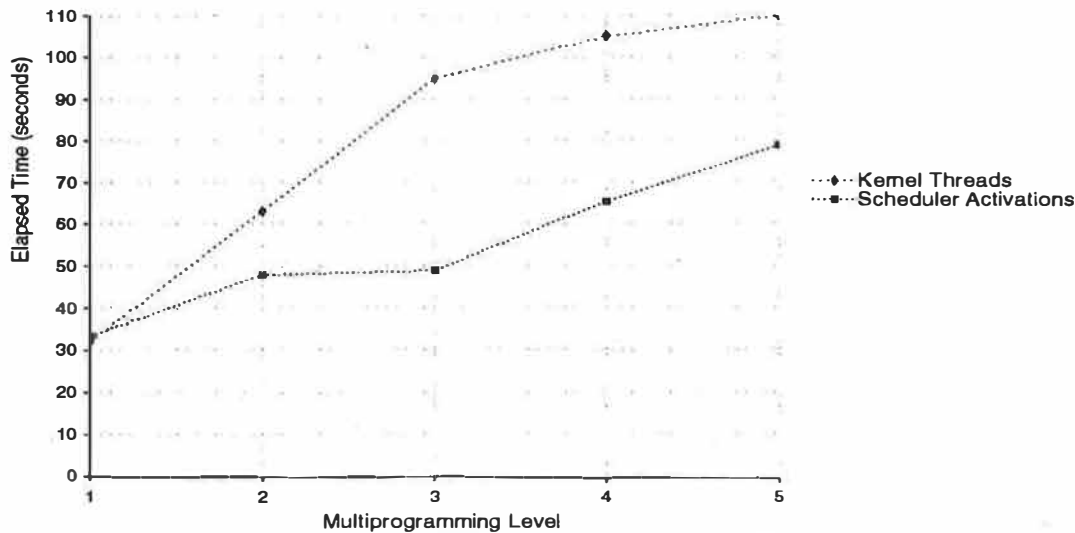


Figure 3: Multiprogrammed Performance of *Gravity*

performance in the presence of competition for processors. Specifically, *Gravity* — which exhibits a “phase” structure common to many scientific programs — can exhibit poor performance if its threads do not reach their internal barriers (between phases) in a timely fashion. Without scheduler activations, scheduling decisions are made by the kernel, and the application therefore cannot guarantee that working threads are given preferential access (over idle threads) to processors. Conversely, when scheduler activations are used, the tasks’ thread schedulers have complete control over which threads are allowed to run, and can arrange a schedule that provides good results.

## 5 Summary

We have incorporated scheduler activations into the Mach operating system, and have modified CThreads to take advantage of this support. The result is a version of Mach in which applications using our CThreads library run with high efficiency, and with the desirable functionality of kernel threads in the presence of system events such as page faults, I/O, and processor preemption. Our implementation favored expediency, Mach compatibility, and portability:

- **Expediency.** We successfully leveraged off existing mechanisms. Examples of this are our use of continuations, processor sets, and kernel threads. We did not use all possible existing mechanisms, however. To implement notifications, we use kernel to user-level upcalls instead of Mach’s message facility; this was a personal choice of expediency. It also cut through some layers of IPC code, perhaps gaining performance.

We emphasize that the interface as described here is not the final word on how best to present scheduler activations as a kernel mechanism. Our interface implements the functionality that any scheduler activations kernel must, but it remains an unanswered question whether the interface is more efficiently implemented via Mach’s IPC system.

- **Backward compatibility.** Our modified kernel is entirely backward compatible with the unmodified Mach kernel. User-level programs and the kernel both behave exactly as before until a task turns on scheduler activations. Even when scheduler activations tasks are running, our use of the processor

set mechanism ensures that other tasks can be isolated from their behavior. Finally, applications can enjoy improved functionality without being rewritten; since the thread system interface is unchanged, applications need only be recompiled and linked to the new thread system that encapsulates event management.

- **Portability.** This issue must be addressed with respect to both the design and the implementation of our system. Since scheduler activations are largely oriented towards shared-memory multiprocessor machines, our design reflects this bias. It can nonetheless be mapped to uniprocessor machines quite easily. The main difficulty on such machines is that the (single) processor must necessarily be time-shared, rather than space-shared as are the (multiple) processors on our Sequent. Decisions regarding the handling of processor allocation in such an environment are necessary (e.g., notifications regarding processor addition and preemption could simply be suppressed). The design of the remaining mechanisms – for performing the other notifications, for example – should be directly portable to uniprocessor architectures. Initial experience [Sav] suggests that this assertion is not overly optimistic.

The portability of the implementation hinges on the degree to which code is processor-dependent. The lone processor dependency in the (added) kernel code is the routine for building upcall stacks, which inherently relies on processor-specific information regarding stack layout and trap/exception context format. Similarly, our changes to CThreads have not affected its processor-dependence, which remains hidden in the context switch, thread startup and spinlock primitive code. Our upcall handlers have some processor dependencies, but these are also hidden via judicious use of processor-specific header files.

Our implementation should provide an effective testbed for future experimentation with scheduler activations, and a useful model for future implementations. We hope that as experience with scheduler activations grows, their impact as a structuring mechanism for future kernels can be more fully understood and exploited.

## 6 Acknowledgments

We would like to thank Brian Bershad for structuring suggestions which helped us integrate scheduler activations into Mach as smoothly as possible. Brian Bershad and others at CMU also helped us get Mach 3.0 running on our Sequent. David Black and Philippe Bernadat from OSF for helping us with some difficulties with the kernel. Discussions with Tom Anderson regarding his implementation provided useful insights. David Kays and Thu Nguyen were involved in several aspects of this project. Sape Mullender gave valuable advice regarding an initial uniprocessor version of this work, done by Vaswani and Peter Bosch at the University of Twente. Jeff Chase provided helpful comments on this paper.

## References

- [ABLL92] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism. *ACM Transactions on Computer Systems*, 10(1):53–79, February 1992. Also appeared in *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, October 1991.
- [ALBL91] Thomas E. Anderson, Henry M. Levy, Brian N. Bershad, and Edward D. Lazowska. The Interaction of Architecture and Operating System Design. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, April 1991.
- [ALL89] Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy. The Performance Implications of Thread Management Alternatives for Shared Memory Multiprocessors. *IEEE Transactions on Computers*, 38(12):1631–1644, December 1989.
- [BH86] J. Barnes and P. Hut. A Hierarchical  $O(n \log n)$  Force-Calculation Algorithm. *Nature*, 24:446–449, 1986.

- [Bla90] David L. Black. Scheduling Support for Concurrency and Parallelism in the Mach Operating System. *IEEE Computer*, 23(5):35–43, May 1990.
- [BLL88] Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. PRESTO: A System for Object-oriented Parallel Programming. *Software – Practice and Experience*, 18(8):713–732, August 1988.
- [BMVL92] Paul Barton–Davis, Dylan McNamee, Raj Vaswani, and Edward D. Lazowska. Adding Scheduler Activations to Mach 3.0. Technical Report UW-CSE-92-08-03, Department of Computer Science and Engineering, University of Washington, August 1992.
- [BRE92] Brian N. Bershad, David D. Redell, and John R. Ellis. Fast Mutual Exclusion for Uniprocessors. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1992.
- [CD88] Eric C. Cooper and Richard P. Draves. C Threads. Technical Report CMU-CS-88-154, Department of Computer Science, Carnegie-Mellon University, June 1988.
- [DBRD91] Richard P. Draves, Brian N. Bershad, Richard F. Rashid, and Randall W. Dean. Using Continuations to Implement Thread Management and Communication in Operating Systems. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, October 1991.
- [Dea93] Randall W. Dean. Using Continuations to Build a User-Level Threads Library. In *Proceedings of the Third USENIX Mach Symposium*, April 1993. This issue.
- [FL89] John Faust and Henry M. Levy. The Performance of an Object-Oriented Thread Package. In *Proceedings of the 4th ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, October 1989.
- [GDFR90] David Golub, Randell W. Dean, Alessandro Forin, and Richard F. Rashid. Unix As An Application Program. In *Proceedings of the Summer USENIX Conference*, June 1990.
- [Her91] Maurice Herlihy. Wait-Free Synchronization. *ACM Transactions on Programming Languages and Systems*, 11(1):124–149, January 1991.
- [Int87] Intel Corporation. *i386 Microprocessor Programmer's Reference Manual*, 1987.
- [LT88] Tom Lovett and Shreekanth Thakkar. The Symmetry Multiprocessor System. In *Proceedings of the 1988 International Conference on Parallel Processing*, August 1988.
- [MSLM91] Brian D. Marsh, Michael L. Scott, Thomas J. LeBlanc, and Evangelos P. Markatos. First-Class User-Level Threads. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, October 1991.
- [MVZ90] Cathy McCann, Raj Vaswani, and John Zahorjan. A Dynamic Processor Allocation Policy for Multiprogrammed, Shared Memory Multiprocessors. Technical Report 90-03-02, Department of Computer Science and Engineering, University of Washington, March 1990. To appear in *ACM Transactions on Computer Systems*.
- [RBF<sup>+</sup>89] Richard Rashid, Robert Baron, Alessandro Forin, David Golub, Michael Jones, Daniel Julin, Doug Orr, and Richard Sanzi. Mach: A Foundation for Open Systems. In *Proceedings of the Second IEEE Workshop on Workstation Operating Systems*, September 1989.
- [Sav] Stefan Savage. Personal communication. February, 1993. `finger savage@cs.cmu.edu`.
- [TG89] Andrew Tucker and Anoop Gupta. Process Control and Scheduling Issues for Multiprogrammed Shared Memory Multiprocessors. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, December 1989.
- [TSS88] Charles P. Thacker, Lawrence Stewart, and Edward Satterthwaite, Jr. Firefly: A Multiprocessor Workstation. *IEEE Transactions on Computers*, 37(8):909–920, August 1988.

# Using Continuations to Build a User-Level Threads Library

Randall W. Dean

*School of Computer Science  
Carnegie Mellon University  
Pittsburgh, Pennsylvania 15213*

(412) 268-7654

Internet: rwd@cs.cmu.edu  
Fax: (412) 681-5739

## Abstract

We have designed and built a user-level threads library that uses *continuations* for transfers of control. The use of *continuations* reduces the amount of state that needs to be saved and restored at context switch time thereby reducing the instruction count in the critical sections. Our multiprocessor contention benchmarks indicate that this reduction and the use of *Busy Spinning*, *Busy Waiting* and *Spin Polling* increases throughput by as much as 75% on a multiprocessor. In addition, flattening the locking hierarchy reduces context switch latency by 5% to 49% on both uniprocessors and multiprocessors. This paper describes the library's design and compares its overall performance characteristics to the existing implementation.

## 1. Introduction

Mach 3.0 [8] has been available since 1990 for anonymous FTP as a platform for building operating systems and embedded applications. To support this development environment, a library that implements multiple threads of control and mutual exclusion is provided. In Mach 2.5 [1] C-Threads [6] is the library that supplies these needs. The Mach 2.5 implementation maps user threads one-to-one with Mach kernel threads resulting in extensive usage of kernel data structures and expensive kernel context switches. The old Mach 3.0 implementation solves the one-to-one mapping problem by multiplexing user threads onto equal or fewer kernel threads. The old implementation uses a complex locking hierarchy suitable for multiprocessors that requires a small state machine inside the context switch critical section. Combined with the need to acquire multiple locks and save full user register state, this approach results in a relatively high latency for user context switching.

The need to save full register state at context switch time also affects throughput on multiprocessors. Regardless of how finely grained the locking hierarchy is, it is necessary to hold a single central lock during a context switch to protect the blocking computation from being resumed by another thread. The time needed to save and restore the user registers becomes the lower bound on critical section size.

We have applied *continuations* to C-Threads to address the effects on latency and throughput caused by long critical sections. *Continuations* have existed in the programming language community as an abstraction for describing a future computation for many years [14]. They have also been used in the Mach 3.0 kernel [7] as a basis for control transfers between execution

contexts. Their use in describing control transfers in the kernel reduced context switching times and kernel stack utilization.

In C-Threads, continuations have not only directly reduced critical section size, but have enabled further optimizations. The direct effect of applying continuations is a reduction of the number of words saved and restored at context switch time. By building a continuation when a potentially blocking C-Thread call is made, the thread state needed at context switch time can be described with only 2 words. The building of the user continuation can be done upon entering the C-Threads library so no lock needs to be held. This reduction in critical section size makes it possible to flatten the locking hierarchy which further reduces the size of the critical section. The net result is a reduction in latency of 5% to 49% on our uniprocessor context switch benchmark. These changes along with *Busy Waiting*, *Busy Spinning*, and *Spin Polling* increase throughput by as much as 75% as measured by our multiprocessor contention benchmark.

Considerable research has been done in the area of user-level thread packages. Of particular relevance to the work described in this paper is the work on scheduler activations [2] [3] and adaptive spinning strategies [12]. We believe there would be synergistic benefits from the integration of these efforts.

Section 2 gives an overview of the C-Threads library interface and describes the relevant internal interfaces and implementations. Section 3 describes how continuations were implemented and used with the new version of C-Threads. Section 4 describes the optimizations used in implementing the new version. Section 5 compares latency and throughput in the old and new versions of C-Threads and shows how certain optimizations effected throughput.

## 2. C-Thread Internals

C-Threads [6] was originally designed as a threading library that transparently used Mach features to supply parallelism. Implementations included a version that used a Mach task for each C-Thread, a version that used a single thread in a single task to supply coroutines, and a version that used one Mach task with a Mach thread per C-Thread. The old Mach 3.0 version of the library multiplexes C-Threads onto equal or fewer kernel threads because of concerns about kernel data structure use such as kernel stacks and kernel context switching times.

The remainder of this section outlines the C-Thread external interface. It also describes enough of the old Mach 3.0 implementation's internal interfaces and structure to provide a basis for understanding the changes made in the new version.

### 2.1. Interface

A detailed description of the C-Threads interface can be found in the CMU technical report by Cooper and Draves [6]. This section briefly reviews the interface described in that document. Where additions to the interface were made, motivation and a more detailed description of the calls is given.

### 2.1.1. Threads

C-Threads are created and managed with the following calls:

- **pthread\_t pthread\_fork(pthread\_fn\_t func, void \*argument)** creates a C-Thread that executes function **func** with argument **arg**.
- **void pthread\_exit(void \*status)** exits the current C-Thread returning **status**.
- **void \*pthread\_join(pthread\_t thread)** waits for C-Thread **thread** to terminate and returns the exit **status** of that thread.

### 2.1.2. Mutexes

C-Threads provides a mutual exclusion primitive called mutex that guarantees at most one holder. Mutexes are manipulated by the following calls:

- **pthread\_lock(pthread\_t m)** Acquires the mutex **m**.
- **pthread\_unlock(pthread\_t m)** releases the mutex **m**.

### 2.1.3. Condition Variables

Condition variables allow one thread to wait for another thread to signal that event or change in state occurred. The following calls handle condition variables:

- **void pthread\_wait(condition\_t c, pthread\_t m)** waits for condition **c** releasing mutex **m** while waiting. When the condition is signaled, the mutex is reacquired before **pthread\_wait()** returns.
- **pthread\_signal(condition\_t c)** signals condition **c** waking up at least one waiting thread.

### 2.1.4. Kernel Threads

The old C-Thread library multiplexes multiple C-Threads onto equal or fewer kernel threads. The best performance is achieved when pre-emptive kernel context switching is minimized without losing parallelism. Two new interface calls were added to allow the application designer to control the number of kernel threads used:

- **void pthread\_set\_kernel\_limit(int limit)** sets the maximum number of kernel threads that can be created. A value of zero indicates that there is no limit.
- **int pthread\_kernel\_limit(void)** returns the current maximum number of kernel threads.

### 2.1.5. Thread Wiring

The C-Threads library includes a mechanism to allow a user to disable C-Threads multiplexing on a per C-Thread basis:

- **void pthread\_wire(void)** binds a C-Thread to its underlying kernel thread.
- **void pthread\_unwire(void)** removes the binding making it possible for multiplexing to occur.

Once a thread is *wired* it becomes possible for the user to make Mach thread calls such as **thread\_priority()**, **thread\_assign()**, or **thread\_info()** on the kernel thread and have the calls only affect the associated C-Thread. Programs can use this facility to run specialized threads at elevated priority, for example, the BSD 4.3 Single Server [8] uses this for threads that interact with kernel devices.

## 2.2. Old Implementations

The old Mach 3.0 implementation draws from the Mach 2.5 coroutine and threads versions. The coroutine version multiplexes multiple C-Threads onto a single Mach kernel thread. This requires machine dependent code for user context switching, per-object queues, and a global queue of runnable threads called the *run\_queue*. With only one kernel thread there can be no contention for the internal data structures so no locking is required in the coroutine version. The threads version binds C-Threads to kernel threads and has as many kernel threads as there are C-Threads. Per-object queues must now be locked because of pre-emptive scheduling, but no machine dependent context switching code or global *run\_queue* is needed since this is all handled by the kernel.

From the Mach 2.5 implementations was taken all of the queues, all of the locking, the multiple kernel threads, and the machine dependent context switching code. With these two sources also came an internal programming layer called *cprocs*. *Cprocs* provides an implementation independent virtual processor layer that is used to implement the higher level C-Threads primitives. The differences between the various C-Threads implementations are hidden within this layer. The remainder of this section will describe interfaces and implementation details that reside below the *cproc* layer.

### 2.2.1. Locking

Locking internal C-Threads data structures cannot be handled with mutexes since they are implemented by the library. Instead, C-Threads does its locking with *spin\_locks*. *Spin\_locks* are implemented using the hardware's native test-and-set, compare-and-swap, or exchange-memory instruction. On uniprocessors that do not supply such an instruction, a kernel-supported software restartable-atomic-sequence is used [4]. In order to minimize bus contention when locking a *spin\_lock*, we implement them as a test followed by the atomic operation. This keeps most of the spinning in the cache.

For each mutex and condition variable there is a queue for C-Threads that are blocked waiting on that object. There is also a global *run\_queue* on which a C-Thread is enqueued when it becomes runnable. Each of these queues is protected by a different *spin\_lock*. The *spin\_lock* guarding the *run\_queue* is called the *run\_lock*.

### 2.2.2. Blocking and Resuming

The routine invoked when a C-Thread blocks is called *cproc\_block()*. This routine does one of two things depending on whether or not the blocking thread is *wired*. If the thread is *wired*, it blocks by calling *mach\_msg()*, waiting for a message to wake it up indicating that it can run again. If the thread is not *wired*, it checks the *run\_queue* for another thread to run. If there is a C-Thread present, it contexts switches to that thread.

If a thread is not present, then the underlying kernel thread must disassociate itself from the blocking C-Thread and become idle. Since the stack that the kernel thread uses in user space is associated with the C-Thread and not the kernel thread, the first thing necessary is to acquire a stack to use. It is not possible to use the blocking C-Thread's stack, since the blocking thread could become runnable and start running on another kernel thread. A pool of idle C-Threads with small stacks is kept for this purpose. A C-Thread is pulled from this pool or created if necessary. The blocking C-Thread then context switches to this idle C-Thread invoking the routine



*cproc\_waiting()*. *Cproc\_waiting()* simply sits in a loop calling *mach\_msg()* waiting for a message to wake it up. When it receives a message, it acquires the *run\_lock* and checks for C-Thread. If a C-Thread is found, its execution will be resumed. If not, the idle thread loops back to *mach\_msg()*.

When an operation such as *condition\_signal()* or *mutex\_unlock()* makes a C-Thread runnable again, *cproc\_ready()* is invoked. This routine inserts the newly runnable thread on the *run\_queue* and wakes up any idle kernel threads.

### 2.2.3. The State Machine

A small state machine is used to keep track of each C-Thread. A thread can be either RUNNING, BLOCKED, or SWITCHING. The first two states have the obvious meaning and are easy to deal with in *cproc\_ready()* and *cproc\_block()*. The third state, SWITCHING, occurs when a thread is in the process of blocking and has released the lock on the queue of the object for which it is waiting.

## 2.3. Evaluation of Old Implementation

While the old Mach 3.0 implementation is a great improvement over the Mach 2.5 implementations by enabling user level context switching, we felt that further enhancements could be made. The old implementation is very aggressive about blocking. Spinning before blocking has been shown to be superior to just spinning or just blocking [12]. Mach 3.0 has better scheduling primitives for synchronization than *mach\_msg()* [5]. Finally, the old Mach 3.0 code is complex and hard to maintain. We created the new implementation to address our concerns with the shortcomings of the old implementation.

## 3. Continuations

A continuation is an object that fully describes a future computation. A continuation can be built, invoked, and passed around as an argument like any other object. The Mach 3.0 kernel has two types of continuations. The first type of continuation describes the user's state and is built when the thread crosses the protection boundary from user space to kernel mode. The second type describes the state that has accumulated since entering the kernel and is built when the thread blocks inside the kernel.

As with the kernel implementation, there are two distinct continuations in the new version of the C-Threads library. A continuation that describes the user's computation at the time a C-Thread routine is called is built upon entrance to the library. We call this the *User Continuation*. The other continuation present in this implementation is used internally to describe the computation that should occur when a C-Thread is resumed after blocking. We call this the *Internal Continuation*.

We have two different continuations because, like the kernel, the computations they describe are so different. The *User Continuation* describes the user state that has accumulated up to the point of a C-Threads operation being invoked. This is potentially large and difficult to examine, because it resides on the user's stack. Furthermore, the user's state is machine dependent. We build the *User Continuation* to encapsulate this machine dependent state as a machine independent object. The *Internal Continuation* is a small, easily examined, easily manipulated, machine independent object consisting of a function pointer and an argument.

### 3.1. Building and Invoking Continuations

The kernel has a distinct protection boundary that makes it easy to build and invoke user continuations when the boundary is crossed. To make it possible for C-Threads to easily build user continuations, we supply a wrapper to potentially blocking C-Thread calls that acts much like the user-to-kernel protection boundary crossover. Building the *User Continuation* is done without holding any locks. The continuation is built by saving all the callee saved user registers on the current stack and saving the resulting stack pointer. Upon exit from the C-Threads library, this continuation is invoked. This entails setting the stack pointer to that saved at continuation creation, restoring the user's registers, and returning to the user routine which originally called C-Threads.

With the user's state encapsulated in the *User Continuation*, all that is needed to describe the state of the library is a single function pointer and argument. The *Internal Continuation*'s two words are saved inside the *User Continuation* in space reserved for this purpose. When this thread is resumed and its continuation is invoked, the function is called with the specified argument. The thread's stack pointer is derived from the address of the previously saved *User Continuation*.

### 3.2. The *filter* Mechanism

To centralize the manipulation of continuations, a single object called a *filter* is used. All manipulations of continuations within C-Threads are done through this object. A *filter* is a per-thread specified function that is invoked each time the manipulation of a continuation is necessary. A *filter* takes as a parameter the type of manipulation necessary. These are:

- **FILTER\_USER\_BUILD:** This is invoked by the wrapper of a potentially blocking C-Thread routine. It calls the real function, which is passed to it as an argument.
- **FILTER\_INTERNAL\_BUILD:** This is invoked when a C-Thread actually blocks. It is passed the function and argument that must be invoked when the C-Thread continues. It calls the **FILTER\_INTERNAL\_INVOKE** of the C-Thread that is being resumed.
- **FILTER\_INTERNAL\_INVOKE:** This is invoked by a blocking C-Thread from **FILTER\_INTERNAL\_BUILD**. It invokes the routine and argument saved in the *Internal Continuation*.
- **FILTER\_USER\_INVOKE:** This is invoked upon leaving C-Threads after entering with **FILTER\_USER\_BUILD**. It invokes the *User Continuation*.
- **FILTER\_PREPARE:** This is invoked when C-Threads creates a new C-Thread. It is passed a routine and argument like **FILTER\_INTERNAL\_BUILD** that will be invoked when the created C-Thread begins running. This is essentially the same as **FILTER\_INTERNAL\_BUILD** without the invocation of **FILTER\_INTERNAL\_INVOKE** upon exit.

Filters can be defined outside of C-Threads. The *filter* described in the section, the *default\_filter*, is the *filter* specified for a thread when it is created. The following two calls allow the *filter* for a C-Thread to be examined and changed:

- **cthread\_fn\_t cthread\_filter(cthread\_t thread)** returns the current filter for C-Thread *thread*.

- `void cthread_set_filter(cthread_t thread, cthread_fn_t func)` sets the current *filter* for C-Thread `thread` to `func`.

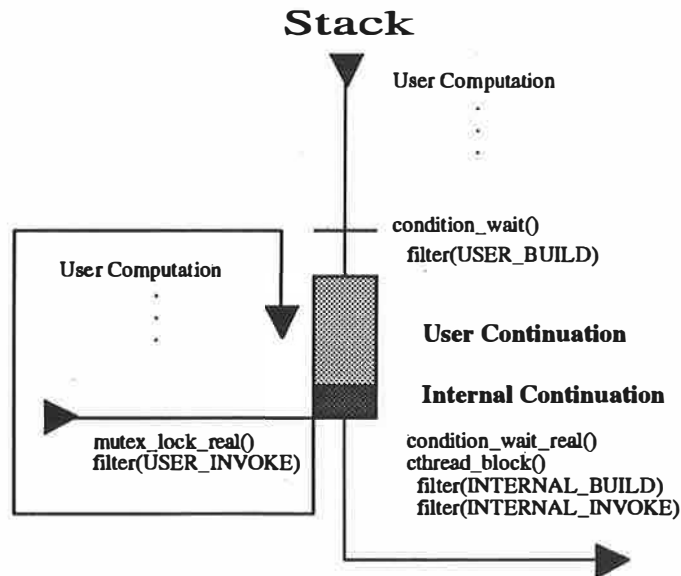


Figure 3-1: Example of user blocking with `condition_wait()`

### 3.3. Example of Continuation Use

Figure 3-1 shows the path taken as a user computation blocks with `condition_wait()`. The user calls `condition_wait()` which is a wrapper that invokes the *filter* to build the *User Continuation*. The *User Continuation* and space for the *Internal Continuation* appear next on the stack as built by the *filter*. The real `condition_wait()` is then called and the C-Thread blocks. When it blocks it invokes the *filter* again to build the *Internal Continuation*. This is saved in the space previously allocated for it when the *User Continuation* was built. The blocking thread then invokes the *Internal Continuation* of the C-Thread to which it is switching, removing the kernel thread from the blocking C-Thread and leaving it blocked.

When the blocked thread is resumed by another thread invoking the blocked thread's *Internal Continuation*, `mutex_lock()` is called to reacquire the mutex released when it blocked. Assuming this succeeds, the *User Continuation* is invoked and the user's computation continues with the stack back where it originally was when `condition_wait()` was first called.

## 4. Optimizations

Continuations enabled a number of changes to the new implementation of C-Threads. These include a flattening of the locking hierarchy and various sorts of spinning when the computation is no longer able to make progress. The optimizations that require spinning occur only on multiprocessors and assume either sequentially consistent [13] or processor consistent [9] memory. This section describes these changes and the motivations behind them.

#### 4.1. Locking Techniques

The old implementation uses separate *spin\_locks* to protect the *run\_queue* and the internal queue of each mutex and condition. The old implementation also requires a state machine to keep track of C-Thread state as threads are enqueued and dequeued and locks are acquired, released and reacquired. By using only a single *spin\_lock* to protect all the queues and the internal state, we eliminate the need for the SWITCHING state of the state machine. Once the SWITCHING state is removed, there is no longer a reason to have the state machine at all.

While the use of only one *spin\_lock* creates a single point of contention, it also makes it easier to add contention reducing optimizations. With this change and the use of continuations, the critical section of the context switch path actually becomes shorter than in the old implementation.

#### 4.2. Busy Spinning and Busy Waiting

*Busy Spinning* is defined as spinning until some global state changes when there may be other work that the processor could accomplish. Spinning before blocking has been shown to be superior to only spinning or only blocking [12]. The new implementation spins for a constant number of iterations before taking action to yield the processor. The number of iterations to spin is currently a static, machine-dependent value determined at compile time. *Busy Spinning* currently occurs whenever a *spin\_lock* must be taken and in *mutex\_lock()* idle threads. We only *Busy Spin* on multiprocessors, since spinning would guarantee the obstruction of work on uniprocessors.

In *mutex\_lock()* the thread spins attempting to acquire the mutex. If the mutex is acquired, the thread returns by invoking the user continuation. Otherwise, after spinning for the maximum number of iterations and not acquiring the mutex, the thread then attempts to acquire the *run\_lock*, queue the mutex, and block.

An idle thread spins on the *run\_queue* without holding the *run\_lock* waiting for a C-Thread to appear that can be run. If this occurs, the idle thread attempts to acquire the *run\_lock*, dequeue a C-Thread, and switch to it. After spinning for the maximum number of iterations and not having a C-Thread appear, the idle thread depresses its priority and begins again [5]. This priority depression yields the processor to any runnable kernel thread. When there are no runnable kernel threads except the depressed thread, the system automatically aborts the priority depression and resumes the idle thread. We call this depressed priority spinning or *Busy Waiting*. C-Threads that are *wired* also use *Busy Waiting*.

Unfortunately, *Busy Waiting* will not work perfectly on a machine with other tasks running if these tasks have compute bound threads. Once an idle thread depresses its priority, it would never run again if there were another thread on the system running. To address this, we use a timeout when we depress priority in the general *spin\_lock* case so that starvation does not occur. For the idle thread we keep a queue of the kernel threads currently idle, and when we insert a thread on the *run\_queue*, we call *thread\_depress\_abort()*. This aborts the priority depression and returns the thread to its original priority.

The old Mach 3.0 C-Threads implementation uses *mach\_msg()* instead of *Busy Waiting* in the idle thread. When the *run\_queue* is empty the idle thread waits to receive a message indicating

that a C-Thread has been inserted in the queue. This disadvantage of this approach is that it is not possible for the idle thread to act on the newly runnable C-Thread until it receives a message. The C-Thread languishes in the *run\_queue* for the duration of a message send followed by a receive. It is also the case that sending a message and receiving a message are considerably costlier operations than depressing priority and aborting it.

### 4.3. Spin Polling

In both of the previously described specific cases of *Busy Spinning*, one of the termination conditions of spinning results in attempting to acquire the *run\_lock*. Since the *run\_lock* is the only *spin\_lock* protecting state used internally by C-Threads, in a finely grained parallel application contention is likely. With *mutex\_lock()* this occurs after a failure to acquire the mutex. With an idle thread, it occurs when a C-Thread appears in the *run\_queue*. In both of these cases, instead of only spinning attempting to acquire the *run\_lock*, we spin both attempting to acquire the lock and checking that the condition that required its acquisition is still true. If after a fixed number of iterations the thread neither acquires the *run\_lock* nor determines that the thread no longer needs to acquire it, the thread yields the processor by depressing its priority. This optimization is called *Spin Polling*.

### 4.4. Continuation Recognition

The *default\_filter* uses continuation recognition, an optimization first proposed and used in the Mach 3.0 kernel [7]. When the *FILTER\_INTERNAL\_BUILD* call of the *default\_filter* prepares to call *FILTER\_INTERNAL\_INVOKE* of the thread it is activating, it checks to see if the *filter* of the resuming thread is the *default\_filter*. If it is, then the *default\_filter* can manipulate the *Internal Continuation* of the newly activated thread to be resumed directly, without needing to make the generalized *FILTER\_INTERNAL\_INVOKE* call.

Another form of continuation recognition occurs within *condition\_signal()*. The continuation of a thread that is being continued by *Condition\_signal* is *mutex\_lock\_solid()*. By examining the *Internal Continuation* and determining the actual mutex that the blocked thread will be attempting to acquire, *condition\_signal()* can make a better choice about which queue to insert the thread into. If the mutex is currently held, then the thread is inserted into the mutex queue where it will later be resumed when the mutex is released. If the mutex is not held, then the thread is inserted into the *run\_queue* with the expectation that it will succeed in acquiring the mutex when resumed.

## 5. Performance

This section describes two benchmarks designed to measure throughput and latency. We first compare the old and new implementations and then examine the effects of the previously described implementation decisions and optimizations. Two platforms are used to show these results: a Decstation 5000/20 Workstation with a 20MHz R3000 [11] and a Sequent Symmetry with twenty 20MHz 80386's [10].

## 5.1. Latency

To show the latency of blocking operations, we use a simple PingPong test that has two C-Threads context switch back and forth via a condition variable. Three versions of this program were created: one that sets the kernel thread limit to two threads, a second that sets the limit to one thread, and a third that does not limit the number of kernel threads, and *wires* each thread. Figure 5-1 shows the results of these tests.

	Decstation			Sequent		
Version	Old	New	Ratio	Old	New	Ratio
1 Thread	146 $\mu$ s	75 $\mu$ s	1.95	151 $\mu$ s	133 $\mu$ s	1.14
2 Threads	578 $\mu$ s	75 $\mu$ s	7.71	268 $\mu$ s	146 $\mu$ s	1.84
Wired	503 $\mu$ s	256 $\mu$ s	1.96	1225 $\mu$ s	447 $\mu$ s	2.74

Figure 5-1: PingPong benchmark results

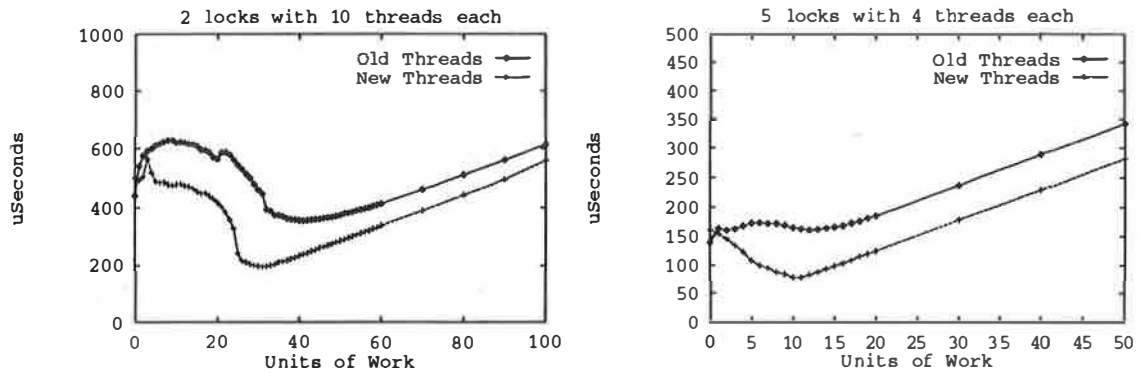
Running the PingPong test with one kernel thread measures the latency of context switching. This is labeled "1 Thread" in Figure 5-1. As can be seen, nearly a two-fold improvement is seen on the Decstation. The latency on the Sequent improves also, but not by as much. This relative difference in improvement might be attributable to differences in the respective memory systems.

Running the PingPong test with the threads *wired* measures the cost of the synchronization primitive used by the underlying kernel. As described earlier, the old implementation uses *mach\_msg()* for synchronization, while the new implementation uses thread priority depression. On the uniprocessor Decstation, the differences in synchronization costs result in nearly a factor of two performance improvement. The multiprocessor Sequent benefits more since explicit synchronization is not actually necessary to resume the second thread. With the old implementation, the second thread will not run until it receives a message from the first. This results in dead time during each iteration equivalent to the cost of a message send and message receive. Since the new implementation uses *Busy Waiting* in *wired* threads, the second thread will begin running immediately upon the condition being signaled. On a multiprocessor, the execution of the first thread's *thread\_depress\_abort()* overlaps the progress being made in the second thread. There is no equivalent cost to the message receive.

Running the PingPong test with two unwired kernel threads adds a way for progress to be made by allowing user-level context switching to occur. On the Decstation, a quantum, the amount of time before a pre-emptive kernel context switch will occur, is 15625 $\mu$ s. It is possible for a kernel thread to execute as many as 200 context switches before the another kernel thread will be pre-emptively scheduled. In the PingPong test, when the second kernel thread does run, if it cannot make progress, it yields the processor back to the first. If it can make progress, it will continue processing user-level context switches until another kernel context switch occurs. In contrast, the old implementation, with its higher latency and higher synchronization costs, can execute at most 100 user context switches between kernel context switches.

## 5.2. Contention Comparisons

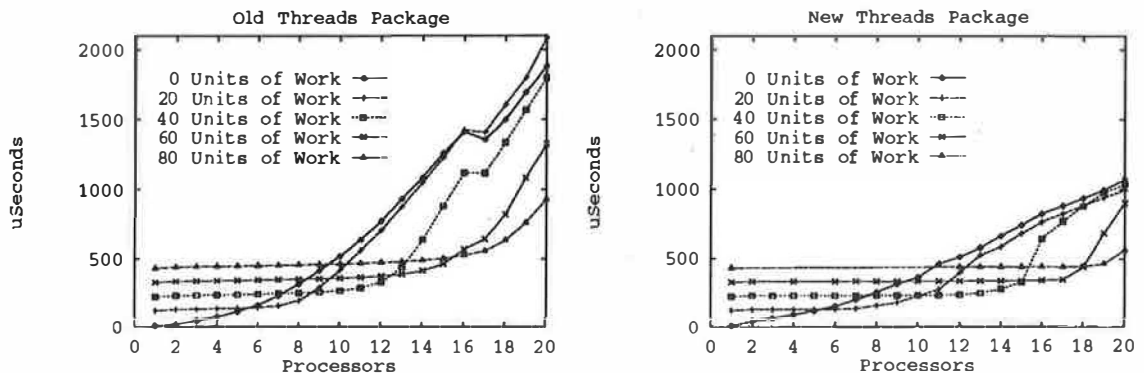
To measure throughput, we ran a benchmark that generates lock contention and measures how long it takes multiple threads to make progress while contending for these locks. The contention benchmark creates a set of locks and associates a pool of threads which each lock. Each thread does some number of units of simulated work then acquires its lock, increments a common counter, and releases the lock. On the Sequent one unit of work takes 5.2 $\mu$ s.



**Figure 5-2:** Comparisons of old and new thread packages on the Sequent

Figure 5-2 shows the contention benchmark on the Sequent comparing the old and new implementations with two different lock and thread parameters. If there were no lock contention, these graphs would be straight lines with slopes equal to the duration of one unit of work and intercepts equal to the overhead associated with creating and deleting the threads. As can be seen, the graphs are not straight lines and contention does exist as the amount of work done between acquisitions approaches zero.

The results of this benchmark show the throughput of new implementation is significantly greater than that of the old implementation. The overhead in the new implementation is lower because its primitives for creating and deleting threads are faster. In the old implementation there is a higher threshold of work above which contention is insignificant, which further widens the performance gap between the two implementations. When work approaches zero, the new and old implementations perform about equally well.



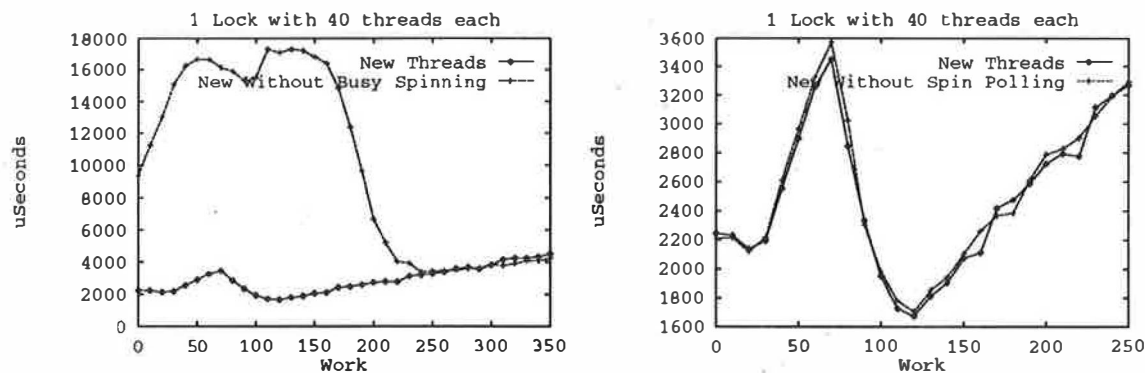
**Figure 5-3:** Old and New versions varying processors and amount of work on the Sequent

To better show the effect of adding processors to a finely grained application, we ran the contention benchmark with one lock and a varying number of threads, each doing a fixed amount of work. Figure 5-3 shows these results for a number of different amounts of work. As can be seen, the new implementation scales considerably better than the old one. If there is no contention, adding a thread that can run on an available processor should have no effect on the execution time of the benchmark. At some number of processors, the amount of time to execute this test begins increasing. With the old implementation and 40 units of work, contention begins

to affect the execution time at approximately 11 threads. The new implementation is not affected until approximately 13 threads. In general, it appears the new implementation can support two more threads without contention than the old one.

### 5.3. Effects of Different Techniques

To examine the effects of *Busy Spinning* and *Spin Polling*, three versions of the new C-Threads implementation were built with one or both of these features disabled. Figure 5-4 shows the contention benchmark run on the Sequent with 40 threads contending for one lock.



**Figure 5-4:** Effects of *Busy Spinning* and *Spin Polling* on throughput on the Sequent

The left graph in Figure 5-4 shows the effect of disabling *Busy Spinning*. The use of *Busy Spinning* provides an order of magnitude performance improvement in throughput under contention. The right graph shows the effect of disabling *Spin Polling*. While disabling *Spin Polling* alone does not make a significant difference, disabling both *Busy Spinning* and *Spin Polling* degrades performance by two orders of magnitude.

To better quantify the effects of these optimizations, measurements were made to examine the relative frequency of events, such as, acquiring the mutex while *Busy Spinning* or the *run\_queue* becoming empty while attempting to acquire the *run\_lock* in the idle thread. The contention benchmark was run on the Sequent using one lock, 40 threads, and 80 units of work. As seen in Figure 5-4, 80 units of work is a point of prime contention. The total number of iterations for this test was 100,000 yielding 4,000,000 total attempts to acquire the mutex lock. Figure 5-5 shows these statistics.

	Count	Percent
Mutex Missed	3,913,671	97.8 of Total
Acquired while <i>Busy Spinning</i>	3,761,246	96.1 of misses
Acquired while <i>Spin Polling</i>	28,007	0.7 of misses
Mutex Blocked	124,418	3.2 of misses
Threads Idled	98,754	79.3 of blocked
C-Thread lost while <i>Spin Polling</i>	33,223	25.2 of waiters

**Figure 5-5:** Statistics for the Sequent with 1 lock and 40 threads

As expected from Figure 5-4, Figure 5-5 shows high contention on the mutex with a 97.8% chance of failing to acquire the mutex on the first attempt with 80 units of work. *Busy Spinning*



accounts for 96.1% of the cases where the mutex was missed. This is consistent with the effects shown in the first graph in Figure 5-4; removing *Busy Spinning* decreases performance by almost an order of magnitude. *Spin Polling* is only beneficial in about 1% of the misses but this translates to 18.4% of the cases where the thread was preparing to block. The remaining 3.2% of the misses result in the thread actually blocking and calling `pthread_block()`. Of the calls made to `pthread_block()`, almost 80% result in the kernel thread becoming idle. *Spin Polling* detects the emptying of the `run_queue` before acquiring the `run_lock` in 25% of the idle threads.

These results show that *Busy Spinning* is extremely important when there is contention on a mutex in order to avoid passing that contention to the `run_lock`. When contention is passed on to the `run_lock`, *Spin Polling* removes the need to acquire the lock in about 1 in 5 cases.

## 6. Conclusions

We have built a C-Threads library with lower latency and faster throughput than our previous implementation. Continuations decreased critical section size and made a number of important performance optimizations possible. The new C-Threads library allows applications to be built that will run well on both uniprocessors and multiprocessors without need to recompile or relink. In addition, the use of continuations, which allows the removal of the locking hierarchy and the state machine, both make the system faster and greatly simplifies the code.

## 7. Acknowledgments

Thanks go to everyone who helped in the work and commented on the paper. These include Brian Bershad, Rich Draves, David Golub, Joanne Karohl, Daniel Stodolsky, and Peter Stout.

## References

- [1] Accetta, M.J., Baron, R.V., Bolosky, W., Golub, D.B., Rashid, R.F., Tevanian, A., and Young, M.W.  
Mach: A New Kernel Foundation for UNIX Development.  
*In Proceedings of Summer Usenix.* July, 1986.
- [2] Anderson, T.E., Bershad, B.N., Lazowska E. D. and Levy, H.M.  
Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism.  
*In Proceedings of the 13th ACM Symposium on Operating Systems Principles.* October, 1991.
- [3] Barton-Davis, P., McNamee, D., Vaswani, R. and Lazowska, E.D.  
Adding Scheduler Activations to Mach 3.0.  
*In Proceedings of the 3rd Usenix MACH Symposium.* April, 1993.  
To appear.
- [4] Bershad, B.N., Redell, D., and Ellis, J.  
Mutual Exclusion for Uniprocessors.  
*In Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems.* October, 1992.
- [5] Black, D.L.  
*Scheduling and Resource Management Techniques for Multiprocessors.*  
PhD thesis, Department of Computer Science, Carnegie Mellon University, July, 1990.
- [6] Cooper, E.C. and Draves, R.P.  
*C Threads.*  
Technical Report CMU-CS-88-154, Department of Computer Science, Carnegie Mellon University, February, 1988.
- [7] Draves, R.P., Bershad, B., Rashid, R.F., and Dean, R.W.  
Using Continuations to Implement Thread Management and Communication in Operating Systems.  
*In Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles.* 1991.
- [8] Golub, D., Dean, R.W., Forin, A., and Rashid, R.F.  
Unix as an Application Program.  
*In Proceedings of Summer 1990 USENIX Conference.* June, 1990.
- [9] Goodman, J.R. and Woest, P.  
The Wisconsin Multicube: A New Large-Scale Cache-Coherent Multiprocessor.  
*In Proceedings of the 15th Annual Symposium on Computer Architecture,* pages 422--431.  
Honolulu, Hawaii, June, 1988.
- [10] Intel.  
*386 Programmer's Reference Manual.*  
Intel, Mt. Prospect, IL, 1990.
- [11] Gerry Kane.  
*MIPS RISC Architecture.*  
Prentice-Hall, Englewood Cliffs, NJ, 1988.

- [12] Karlin, A.R., Li, K., Manassee, M.S., Owicki, S.  
Empirical Studies of Competitive Spinning for Shared-Memory Multiprocessors.  
In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*.  
1991.
- [13] Leslie Lamport.  
How to Make a Multiprocessor Computer That Correctly Executes Multiprocess  
Programs.  
*IEEE Transactions on Computers* C-28(9):241-248, September, 1979.
- [14] Robert Milne and Christopher Strachey.  
*A Theory of Programming Language Semantics*.  
Halsted Press, New York, 1976.



# An Architecture for Device Drivers Executing as User-Level Tasks

---

David B. Golub

*School of Computer Science  
Carnegie Mellon University  
Pittsburgh, Pennsylvania 15213*

Guy G. Sotomayor, Jr.

*IBM \*IX Open Systems  
Boca Raton, Florida 33431*

Freeman L. Rawson III

*IBM \*IX Open Systems  
Boca Raton, Florida 33431*

*The existing suite of device drivers used with the Mach 3.0 microkernel is derived from the BSD device drivers used in Mach 2.5 systems. These device drivers have been widely regarded as temporary and as one of the weakest features of the system. Beginning with the notion introduced by Forin, Golub and Bershad that device drivers can run as Mach user-level tasks, we have created a complete device driver model that separates device access from device operation in such a way that multiple device drivers can cooperate in managing a single piece of hardware. We have also defined and implemented a message-based protocol for invoking device driver services and have extended the Mach microkernel to make user-level device drivers more efficient. All of the principal features of our device driver model have been implemented and demonstrated, and by the time of the Mach Usenix Symposium we will have a complete suite of device drivers and will have removed all of the in-kernel device drivers that we are currently using.*

## 1. Introduction

The existing suite of device drivers used with the Mach 3.0 microkernel[1] is derived from the BSD device drivers used in Mach 2.5 systems. These device drivers have been widely regarded as temporary and as one of the weakest features of the system. They exhibit poor latency characteristics in real-time environments and are linked into the kernel forcing a new kernel to be made to add or to remove them. Finally, they tend to be rather specific to the Unix® environment rather than being completely general for use with any operating system personality. In response to these problems, there have been a number of attempts to develop an alternative device driver model for Mach 3.0. One of the better known efforts is that of Forin, Golub and Bershad[2] where device drivers are run as user-space tasks. Our work takes the notion of executing device drivers as ordinary Mach tasks as its starting point. But we also attempt to solve a number of other problems. Given our goal of supporting multiple, potentially conflicting operating system personalities, we separate access to the hardware from device operation so that multiple device drivers can be loaded and share access to a particular piece of hardware. We also need to support device virtualization in order to provide an adequate level of compatibility for PC DOS and programs built using it. We also want to avoid the creation of more programming environments in the system: this goal serves to reinforce the decision to start with the concept of running the device drivers as user-level tasks. By doing so, we avoid the need to create specialized interfaces inside the kernel for use by device drivers, and we also avoid the need to support a general kernel extension mechanism such as that found in AIX®v3. At the same time our device driver model has to provide reasonable performance and reli-

ability in managing a wide variety of hardware. Although it might be possible to implement device drivers as shared library code executing at user level, it seems very difficult to provide the right level of protection with such a scheme.

One of the standard criticisms of the idea of running device drivers as user-level programs has been that doing so would result in a serious performance loss. Particularly difficult tests of our device driver model are devices that deliver large numbers of small packets asynchronously and have little or no buffering, such as relatively high-speed asynchronous communication ports. Our initial results indicate that we can run device drivers at user level with acceptable performance, but we do not yet have definitive results.

The balance of this paper is organized as follows. In Section 2, we review the criteria that we apply to ensure that our device driver model is complete and answers all of the relevant questions about device drivers. Section 3 describes the naming scheme that we use while Section 4 covers the functions and features of the Hardware Resource Manager. Section 5 discusses the device driver execution environment including the support that we have added to the microkernel to make our device drivers run efficiently and to permit the hardware resource sharing that we provide. Section 6 defines the external interfaces that we use with our device drivers while Section 7 describes the internal organization that we suggest for device drivers written in our model. Section 8 describes the changes that we have made to the microkernel itself to support the device driver model. Section 9 gives our initial experiences with our device driver model, describes the work that we currently have underway and relates what we have done to previous work in the area.

## **2. A Complete Device Driver Model for Mach 3.0**

One difficulty with device driver design and development for most operating system environments is that the system's device driver model does not deal with all the problems that must be faced in actually writing a driver. In order to avoid this trouble, we carefully identify each of the five aspects of device driver implementation and specify how each is to be handled in our model. The first aspect of a device driver is naming: how are resources named and addressed? This includes both hardware and the device drivers themselves. The second is hardware access: how do device drivers gain control and manage access to the hardware that they are to control? This problem is especially acute in an environment where different programs have radically different ideas about device drivers such as one finds in an multiple personality environment that includes PC DOS. The third part of a device driver model is the execution environment: how do the device drivers execute on the system? The typical answer has been to make them a part or a close relative of the operating system kernel whereas we have followed [2] in making them Mach user-level tasks. The fourth aspect is the external interfaces that the device drivers present: what interfaces are exported to the programs that call the device drivers? In traditional Unix, there were two different sets of the interfaces—one for the character device drivers and one for the block device drivers. Our approach is to allow for more diversity. The fifth and final portion of a device driver model is the definition of the internal structure of the drivers: how are device drivers structured internally? Following [2], we believe that there is a potential for a tremendous amount of common code between drivers for different devices and are using notions borrowed from object-oriented programming in order to organize this code.

## **3. Naming**

The problem of naming in the presence of multiple user-level device driver tasks and a number of other supporting servers such as the default pager and multiple operating system personalities is complex, and our approach is to provide a naming service that is simple enough for device drivers and other primitive services but flexible enough to form the basic naming service for the entire system. The naming scheme uses Mach ports to represent objects and provides an I/O name service. The I/O name service sets itself up during system initialization. It provides services that convert external ASCII string names of objects to Mach port rights and vice versa. Device drivers can register the names that they wish to support and the Mach ports that they represent with the I/O name service. Attributes and attribute values can be associated with a name to allow additional information to be associated with the name that may be useful to other portions of the system in identifying what the name represents.

### 3.1. I/O Names

There are four aspects to a name managed by the I/O name service. The first is the name itself. The name consists of an ASCII string that is used by other servers in the system to find out about the other aspects of the name. The second is a port. This is usually a port to which messages can be sent and represents a service interface provided by the program that placed the name in the name service. The third is an attribute. Attributes allow names to be classified in generic ways. Finally, values can be given to a name's attribute to give very flexible ways of identifying services that have been placed in the name service. For example, **hd0** may have an attribute of **DRIVER\_TYPE** with a value of **DISK** thereby identifying **hd0** as a disk driver. This allows the clients of device drivers to find the drivers by their type or some other attribute rather than by interpreting the content of individual names or by using an arbitrary convention on them.

### 3.2. Protocols Between Device Drivers and the Name Service

The interfaces to the I/O name service are divided into three categories. The first are the manipulation interfaces. These interfaces allow names, attributes and attribute values to be added to, changed in or removed from the name service. The second are the notification interfaces. These interfaces allow clients of the name service to register interest in names, attributes and attribute values. When a name, attribute or attribute value is added to the name service, the client is notified that the item has been added. This permits higher level system functions to discover when new lower level services become available. The final set of interfaces are the query interfaces. These interfaces allow a client to query the name service in several different ways. The queries can return names, ports, attributes or attribute values. The pattern used to form the query can be any of the components of the name maintained by the name service. In the case of names, wild card characters like those supported by Unix shells can be used for broad pattern matching.

## 4. Hardware Access Management

One of the features of our device driver model is its explicit support of multiple device drivers for the same hardware. The Hardware Resource Manager (HRM) is a collection of user-level Mach tasks that cooperate with the microkernel to export I/O resources to device drivers. The microkernel and the HRM encapsulate all of the hardware resources in the system. The HRM assigns types to them, and it has mechanisms for discovering what hardware resources are present on the machine. Device drivers use a message protocol to acquire access to hardware resources from the HRM. When a device driver wants to access hardware that it does not currently have the right to use, it makes a request to the HRM. The HRM then sends a message to the device driver that currently owns the hardware, asking it to relinquish the device. The response may be to give up the device, to promise to give it up on a later request or to refuse. If and when the device is relinquished by its current owner, the HRM sends a message to the requesting device driver that tells it that now can use the hardware.

### 4.1. The Purpose of the HRM

The notion of a Hardware Resource Manager came from several observations. Previous device driver models tended to have no clear mechanism for identifying the hardware resources that were needed by the device driver except those mechanisms provided by individual device drivers. Since there was no clear mechanism for identifying the resources needed, device drivers would make educated guesses regarding the hardware resources present. Due to these *ad hoc* resource identification schemes, device drivers tended to use any resources identified. Collisions between device drivers utilizing the same hardware resources were only prevented by only by placing restrictions on software configuration and by implementing complex, very specific software protocols. As a result, very few systems support multiple concurrent device drivers for the same hardware device.

Another significant problem with the *ad hoc* schemes used in previous device driver models is that the determination of what hardware resources are present is very sensitive to a particular system implementation. This led to device drivers that were not only specific to a particular hardware device but also to the way in which the whole hardware system forces the device driver to perform its resource identification.

These *ad hoc* resource identification mechanisms also tend to cause the device driver model to be fairly static with the device drivers only performing resource identification during initialization as it may not function reliably when the device is in actual operation or it may cause perturbations to the device or to the system itself. The need for a more dynamic device driver model becomes extremely important with the desire for dynamic configuration and the flexibility of some emerging technologies such as PCMCIA.

The HRM solves the resource identification problem by providing a data base to the device drivers that they can query at any time to determine what resources are present. By providing this data base, the HRM moves the identification out of the device drivers themselves, eliminating one source of unnecessary device driver diversity. Device drivers only need to query the data base: establishing the contents of the data base is only dependent upon a particular implementation of a part of the HRM, not on each device driver. As a result, device drivers are more likely to be dependent solely on the device being supported and not on the hardware platform in which the device is being packaged.

The HRM also makes the system more dynamic because its data base can be updated dynamically. Thus, device drivers can determine changes in the hardware configuration that affect them by querying the data base maintained by the HRM.

However, the primary purpose of the HRM is to permit several device drivers to share the same physical set of hardware resources. Since device drivers query the HRM for the hardware resources, it is simple to arrange to share hardware resources by having the drivers request access to the hardware from the HRM prior to using it. The HRM tracks which device driver owns each set of hardware resources and ensures that only one device driver at a time has it. When another device driver requests the same hardware from the HRM, the HRM can request that the current owner give up the resources. With this protocol each device driver knows only that it must request the right to use the hardware from the HRM and that it may receive requests to give it up, but it does not have to know the identities of any of the other device drivers that may be sharing the hardware with it.

There are several reasons why a device driver could be asked to give up the use of a set of hardware resources. The first is that for one reason or another the system has determined that the hardware is no longer present in the system: this may happen, for example, with PCMCIA devices. However, it is more common for a device driver to be asked to release a hardware resource so that another device driver can use it. For instance, a common problem in many systems is the difficulty of supporting virtual screens: the system must provide an environment where multiple applications that are coded assuming exclusive access to the display hardware can execute concurrently. Virtual screens are relatively easy to implement by having a device driver per screen and having the device drivers cooperate using our protocol.

Another problem that occurs sometimes is the result of hardware developers reusing an existing controller to support a second, totally different type of device from the one initially attached. For example, on the PS/2 the floppy disk controller was eventually used to support both the floppy drive and the internal tape backup unit.. This usually requires the original device driver to be re-written to support both device types. With our protocol it is possible to write two different device drivers for the two different devices and have them share the common controller hardware. Each device driver is ignorant of the presence of the other.

Finally, our protocol permits the system to support online diagnostics in a very elegant manner. With the demand for systems to remain functional longer even in the face of partial failures or to be able to diagnose failures without taking the entire system down, the current diagnostic scheme of stand alone diagnostics is no longer viable. By using the HRM, a diagnostic program can obtain the hardware resources needed to run the diagnostics. The standard device driver for the hardware simply shares the hardware resources with the diagnostic program.

## 4.2. Overall HRM Structure

The HRM is not a single program but a collection of programs that manage the various hardware resources that are contained in a system. It is through the HRM that device drivers and other software subsystems gain access to the physical hardware resources.



The programs that compose the HRM fall into one of the following three categories:

- **Name Service** The name service that is used by the rest of the I/O system is the same one that is described here. It was originally used only in conjunction with the HRM. However, its usefulness to the rest of the system causes the name service to remain part of the HRM for historical reasons only. There is only one program in the HRM that provides the name service.
- **Bus Manager** The bus managers provide most of the function of the HRM.. It is the individual bus managers that contain the data bases that describe all of the hardware resources in a system. A bus manager is specific to an individual type of bus. For example, there are different bus managers for the MicroChannel bus and for the SCSI bus. It is the Bus Manager(s) which implement the *Grant/Yield* protocol discussed above which allows other components in the system to gain access to the various hardware resources.
- **Bus Walker** These programs are matched to a specific bus manager. They are responsible for loading the bus manager's data base for bus(ses) that it is to manage. For certain busses the bus walker is a fairly simple program that may read a script to load the bus manager's data base. For other busses, the bus walker may read a collection of files using those to interpret a hardware-prepared data base such as NVRAM on the PS/2 to load the bus manager's data base.

### 4.3. HRM Data Base

The data base that represents all of the hardware resources in the system is distributed in the various bus managers. The data base maintained by the bus managers is non-persistent. Each bus manager keeps its data base in its memory image as a collection of data structures that can be queried and manipulated. The interaction between the various data bases is minimal and is only provided through HRM interfaces.

The data base is organized as a tree. The leaves of the tree represent the various hardware resources that are present in the system. The tree is not of arbitrary depth. Each branch of the tree is of a particular type and only certain nodes can be parents or children of other nodes. The nodes in the tree can be divided into two major categories—those nodes that describe the topology of the system, that is, what resources *exist* in the system, and those nodes that describe the activity of the system, that is what resources are *in use* in the system.

The nodes that represent the activity of the system are structured into subtrees that shadow the portion of the topological nodes for which activity is being represented. It is important to note here that there may be more than one activity subtree for a particular topological subtree. Each activity subtree represents an individual client using the resources represented by the topological subtree. This shadowing permits the bus managers to track the many-to-one relationship between client device drivers and hardware resources.

Each node has a particular type. The following types are defined for topological nodes.

- **Bus Type** This type of node is exported by each bus manager to indicate the type of bus that it supports. Examples might include MicroChannel, ISA bus and SCSI.
- **Bus Object** This type of node is created by a bus manager to indicate the presence of a physical bus of the type that it supports.
- **Association** This type of node is used to group one or more Resource Objects into a single entity. This allows several physical entities to be grouped together into a logical entity.
- **Resource Object** This type of node is used to group one or more Resources into a single entity. This allows several physical resources to be grouped together to form something analogous to a controller.

- **Resource Type** This type of node is used to indicate the type of a resource. It has imbedded in it by the bus manager certain attributes about the resource. Examples include I/O ports, I/O memory and interrupt levels.
- **Resource** This type of node is used to represent a single physical resource. Examples include an interrupt level, a contiguous range of I/O ports and a contiguous range of I/O memory.

The following node types represent the activity in the system. They are created when a client wishes to actually use the resources represented by a topological subtree. When a client creates an *active instance* of a topological subtree, it does so at the top of the subtree. A complete active subtree is created when the node at the top of the subtree is activated.

- **Association Instance** This type of node is used to represent an active instance of an Association node.
- **Resource Object Instance** This type of node is used to represent an active instance of a Resource Object node.
- **Resource Instance** This type of node is used to represent an active instance of a Resource node.

The following diagram illustrates the types of and relationships between the various nodes in the tree representing the HRM data base.

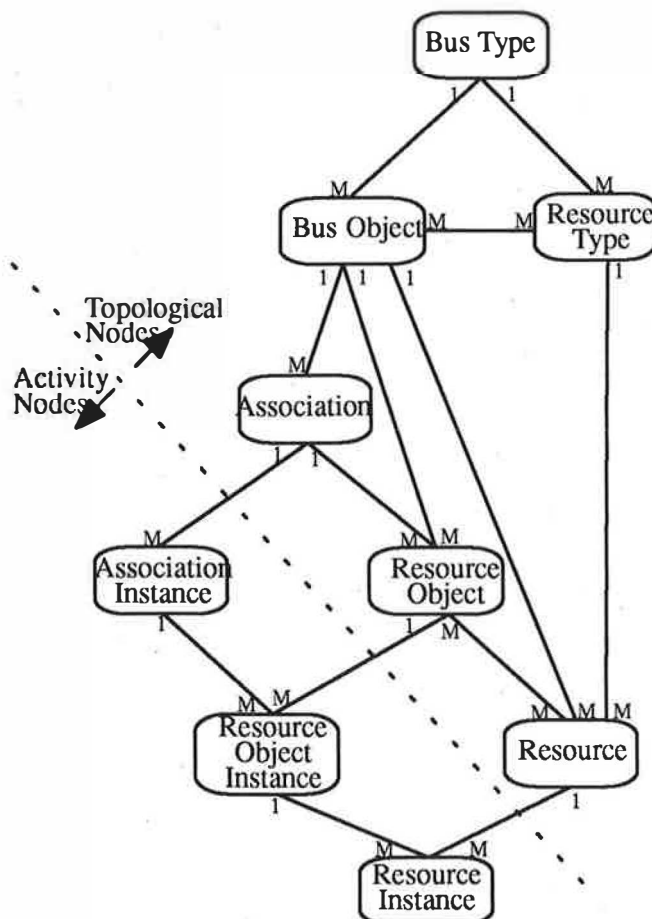


Figure 1: Entity Relationships in HRM Data Base

There are several things that may not be immediately obvious from the diagram. A single Bus Type is maintained by a single bus manager. It is possible for a single executable program to support more than one Bus Type; however, this is treated as multiple bus managers. It is likely that a single bus manager may support multiple Bus Objects, representing multiple physical busses. The Bus Type and Resource Type nodes are defined by the bus manager and are not dependent upon the content of the remainder of the data base. They are always present in the data base since they are placed there directly by the bus manager. All of the other nodes with the exception of the Resource Manager Object are placed into the data base by bus walkers.

#### 4.4. Gaining Access to Resources

Even though an activity subtree has been created in the HRM's data base, the clients of the HRM still do not have access to the actual resources represented by the subtree. In order for a client to obtain access to the physical resources, it must request them. Clients will usually request access to a subtree, that is, an Association or a Resource Object, rather than requesting the individual Resources since it is the Association or Resource Object represents a whole device.

Since there may be several clients attempting to use the same set of physical resources, the HRM resolves the contention by implementing what is called a Grant/Yield protocol. When a client requests a set of Resources, the HRM determines if there is another client accessing the Resources. If no other client is accessing them, the HRM can directly *Grant* the Resources to the requesting client. If there is another client accessing the Resources, the HRM, prior to *Granting* the Resources to the requesting client, must get the other client to *Yield* the Resources.

The *Grant/Yield* protocol is illustrated in Figure 2 through Figure 5. In this example, there are three HRM clients,  $\alpha$ ,  $\beta$  and  $\gamma$ . Client  $\alpha$  represents a client that created an activity subtree starting with the Association *Group*. Client  $\beta$  represents a client that created an activity subtree starting at the Resource Object *Ctlr-B*. Finally, client  $\gamma$  represents a client that created an activity subtree starting at the Resource Object *Ctlr-C*.

Figure 2 shows that at some previous time client  $\gamma$  requested that its instance of *Ctlr-C* be made fully active. This means that client  $\gamma$  has full access to the hardware resources represented by the subtree *Ctlr-C*. This is illustrated by the activity nodes being shaded. It is at this point in time that client  $\alpha$  requests the HRM to give it access to the resources represented by the subtree *Group*.

Figure 3 shows the state of the HRM data base after client  $\alpha$  has requested access to the resources in subtree *Group*. The actions that the HRM has taken to change the state of the data base from that in Figure 2 to what is shown in Figure 3 are described as follows:

1. Client  $\alpha$ 's instance of *Group*'s state is changed to going active. The HRM then starts to activate the subtrees of Resource Object instances that make up client  $\alpha$ 's *Group* instance.
2. Client  $\alpha$ 's instance of *Ctlr-A*'s state is changed to going active. The HRM then starts to activate the subtrees of Resource instances that make up client  $\alpha$ 's *Ctlr-A* instance.
3. Client  $\alpha$ 's instance of *R-e*'s state is changed to going active. The HRM checks the Resource node for *R-e* is checked to see if there is an instance of *R-e* that is already active. There is not, so the HRM changes client  $\alpha$ 's instance of *R-e* to fully active. The exact actions the HRM takes in making an instance fully active are defined by the Resource's Resource Type.
4. The HRM proceeds through the other Resource instances for client  $\alpha$ : there are none since *Ctlr-A* contains only one Resource. Since all of the Resource instances have been visited, the HRM can now mark client  $\alpha$ 's instance of *Ctlr-A* fully active.
5. The HRM has made a Resource Object instance fully active. It now sends a *Grant* message to client  $\alpha$  indicating that the client now has fully access to all of the Resources contained in the Resource Object.

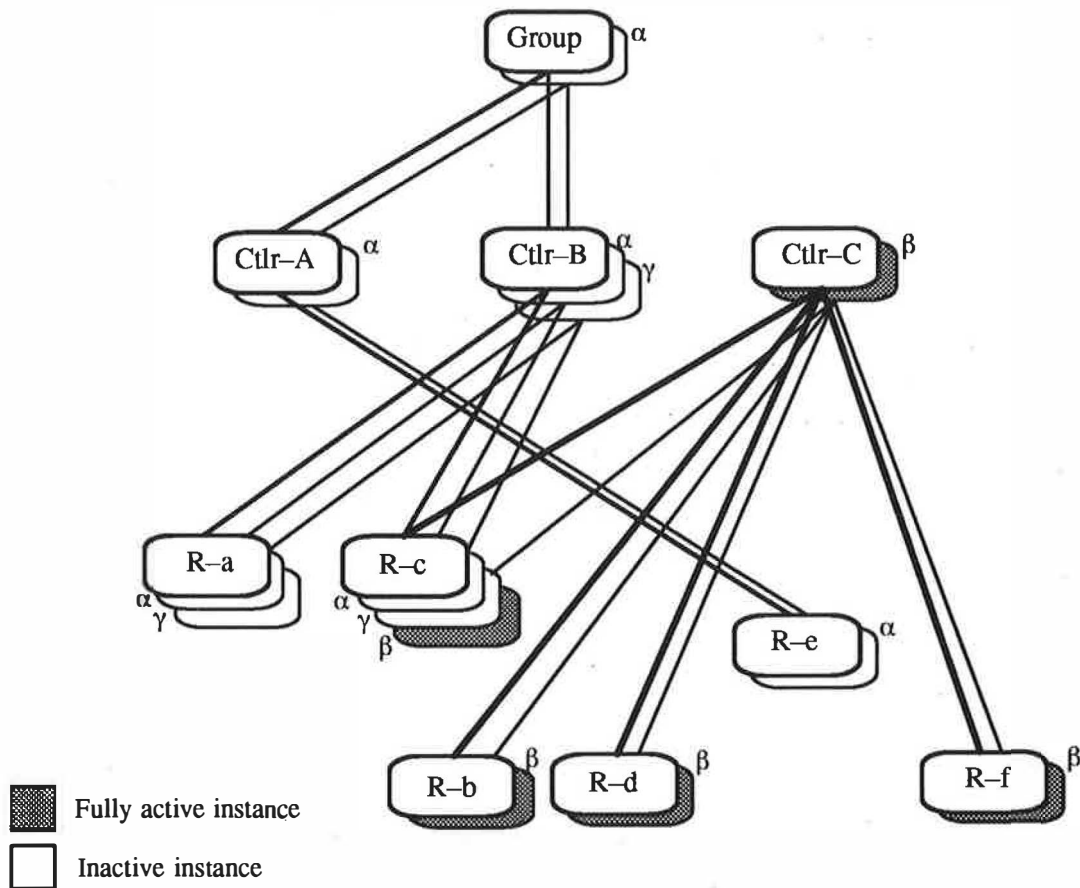


Figure 2: *Grant/Yield* Example – initial state

6. The HRM proceeds to the other Resource Object instance(s) in the Association. In this case there is only one left and that is *Ctrlr-B*. Client  $\alpha$ 's instance of *Ctrlr-B* is changed to indicate that it is becoming active.
7. Client  $\alpha$ 's instance of *R-a*'s state is changed to going active. The HRM checks the Resource node for *R-a* is checked to see if there is a instance of *R-a* that is already active. There is not, so the HRM changes client  $\alpha$ 's instance of *R-a* to fully active. Again, the exact actions the HRM takes in making an instance fully active are defined by the Resource's Resource Type.
8. The HRM now proceeds to go through the other Resource instances for client  $\alpha$ . There is only one remaining and that is the instance for Resource *R-c*. It's state is marked as becoming active. The HRM checks to see if there is another instance of Resource *R-c* that is active. There is, and that is the instance for client  $\beta$ . That instance's state is marked as becoming inactive.
9. At this point the HRM will send a message to client  $\beta$  indicating that it should yield access to the hardware represented by its instance of Resource *R-c*. Upon receipt of this message, the client must save any state that is relevant to the particular resource. Once the client replies to the *Yield* request, it must assume that it no longer has any access to the resource.

Figure 4 illustrates the state of the HRM data base after it completes the processing of the *Yield* reply from client  $\beta$ . The actions that the HRM has taken to change the state of the data base from that in Figure 3 to what is shown in Figure 4 are described as follows:

1. Once the HRM receives the reply to its *Yield* request, it marks client  $\alpha$ 's instance for Resource *R-c* as active.

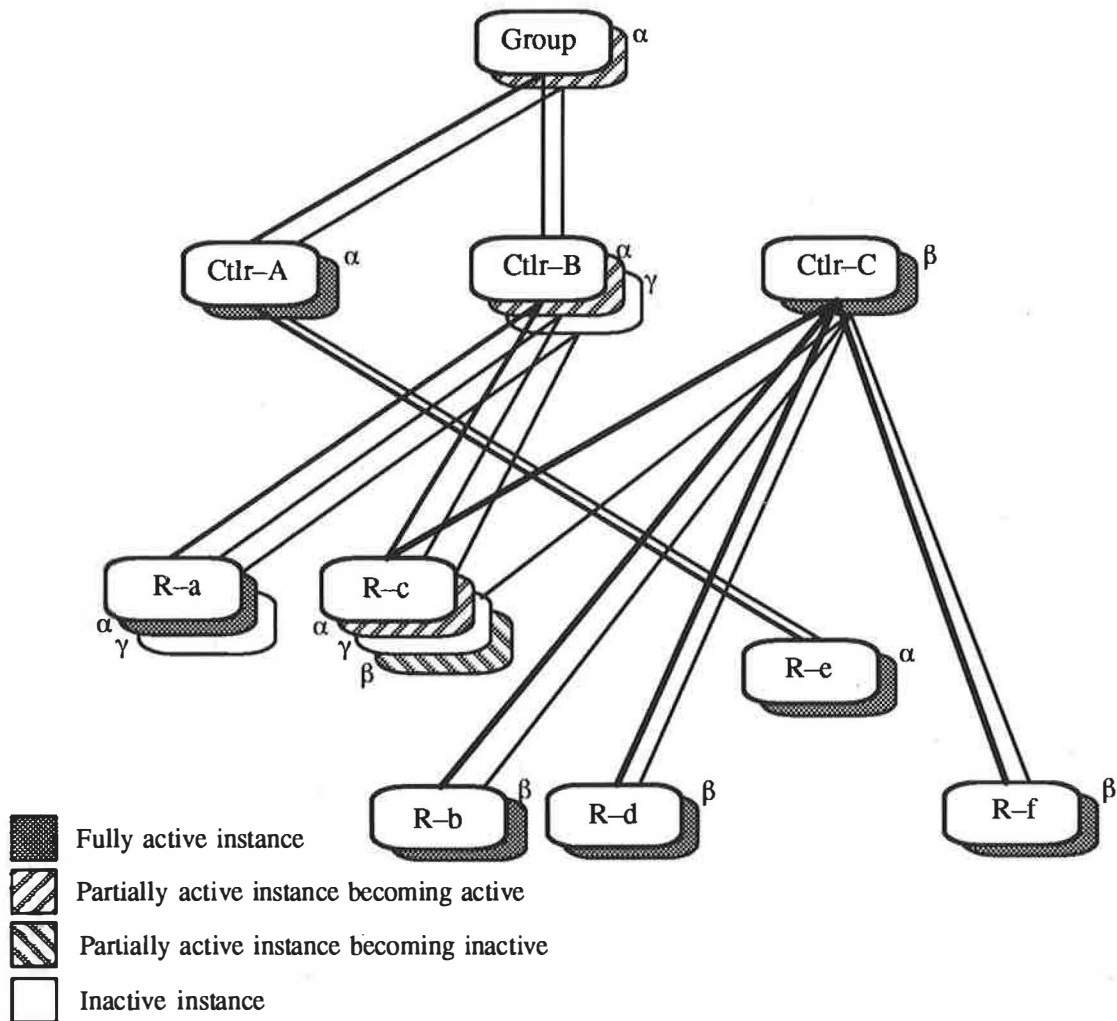


Figure 3: *Grant/Yield* Example – client requests access

2. The HRM proceeds through the other Resource instances for client  $\alpha$ : there are no more since *Ctrl-B* contains only two Resources. Since all of the Resource instances have been visited, the HRM can now mark client  $\alpha$ 's instance of *Ctrl-B* fully active.
3. The HRM has made a Resource Object instance fully active. It now sends a *Grant* message to client  $\alpha$  indicating that the client now has fully access to all of the Resources contained in the Resource Object.
4. The HRM proceeds through the remaining Resource Object instances for client  $\alpha$  that are part of Association *Group*. Since there are no others, client  $\alpha$ 's Association instance for *Group* is now marked fully active. A reply is sent back to client  $\alpha$  in response to the original request message.

It is usually the case that when a client receives a *Yield* request for a Resource that is present in a Resource Object, the client voluntarily yields the remaining Resources that are present in the Resource Object. This allows for simpler tracking of the state of the Resource Object by the client. It is usually more meaningful to the client since it is usually interested in the collection of Resources represented by a Resource Object as a single entity. A client also tends to save the entire state of the hardware when one portion of it is yielded because it may not be possible to obtain the state after the client replies to the *Yield* request from the HRM.

Figure 5 represents the state of the HRM data base once the client  $\beta$  has issued *Yield* requests to give up access to the remaining Resources that it still has access to. The actions that the HRM has taken to

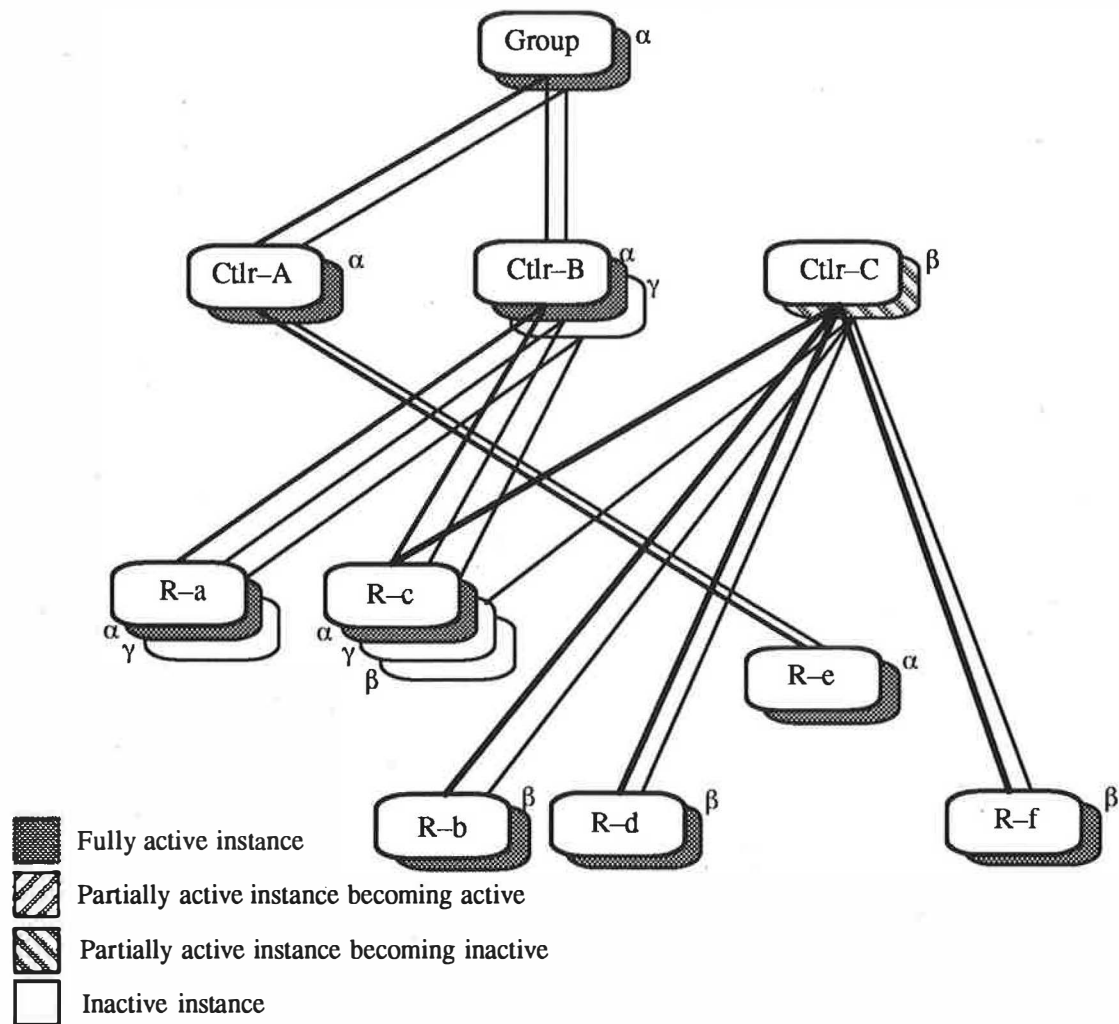


Figure 4: *Grant/Yield* Example – new client granted access

change the state of the data base from that in Figure 4 to what is shown in Figure 5 are described as follows:

1. Client  $\beta$  sends a *Yield* request to the HRM for its *Ctrl-C* Resource Object instance. The HRM receives the request and marks client  $\beta$ 's Resource Object instance for *Ctrl-C* as becoming inactive.
2. The HRM examines each of the Resource instances that correspond to client  $\beta$ 's Resource Object instance for *Ctrl-C* to determine if they are active. For each Resource instance that is found active, the HRM will send a *Yield* message to the client after changing the state of the Resource instance to becoming inactive.
3. When client  $\beta$  responds to a *Yield* message from the HRM, it saves the remaining residual state for the hardware represented by the Resource and replies back to the HRM. When the HRM receives the reply from the client, it marks the state of the Resource instance as being inactive.
4. When all of the Resource instances have been yielded by the client, the HRM marks the state of the Resource Object instance as being inactive. It then sends a reply back to the client indicating that Resource Object instance was yielded.

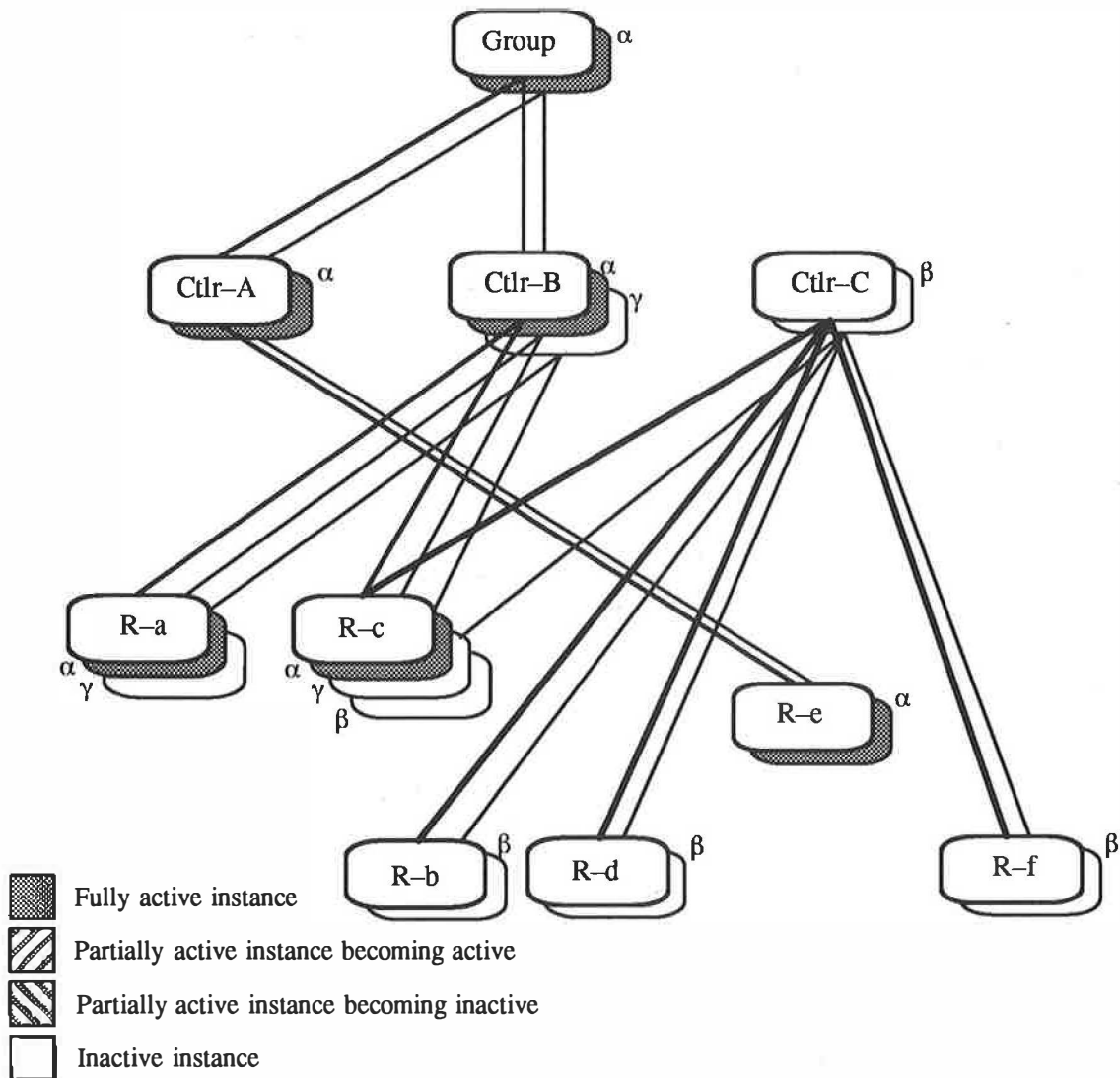


Figure 5: *Grant/Yield* Example – old client yields access

#### 4.5. Resource Discovery

Hardware resource discovery is performed by the bus walkers that are matched to the particular bus managers. When a bus walker runs, it determines in a manner specified by the type of hardware platform that it supports if hardware resources have been added or removed from the system. It does this by determining what hardware is present and matching that against what is contained in the bus manager's data base. The bus walker adjusts the data base to reflect the actual hardware configuration. A bus walker can be run or re-run at any time. This therefore allows the collection of bus manager data bases to be dynamic.

All of the names of the hardware resources contained in the bus manager data bases are placed in the name space managed by the name service. Since the name service supports notification when changes are made to its contents, device drivers and other portions of the system can be notified when new resources appear. This allows device drivers to be informed of changes in the hardware configuration that they may be interested in when the changes occur.

### 5. Execution Environment

The execution environment for device drivers in our model is that of a user-level, stand-alone, personality-neutral Mach task. However, there are some interrupt handling functions that must be performed in

the kernel. We have decided that for efficiency reasons that it is also necessary to have a small portion of the device driver execute off of the interrupt level when a hardware interrupt is presented. The amount of code that is contained in the interrupt handler varies between device drivers but is usually less than 100 machine instructions. The biggest advantage of running the majority of the code at user level is that it gives us a very uniform programming model: all programs that are not personality-specific have the same execution environment. This allows for code reuse, simplicity of development, and ease of testing and debugging.

## 6. External Interfaces

The external interfaces presented by the device drivers in our model are intended to remove many of the limitations of Unix and of the current Mach `device_*` interfaces. They are based on a message protocol. We have defined a hierarchy of device driver classes with more specific device types being arranged into subclasses. We start with an overall device class to support device-independent I/O. Underneath this, there are storage and stream classes with the stream class being subdivided into character, network and console classes. The idea is to make the interfaces as similar as possible among the different types of devices while avoiding the rigidity of the traditional Unix classification into block and character devices.

In defining the external interfaces to device drivers, we have borrowed from object-oriented programming by defining a hierarchy of device driver classes. The interfaces for a particular type of device driver are defined by subclassing the basic classes. Figure 6 shows the class hierarchy that our current device drivers use in their interface definitions. Device drivers typically export several objects. Among

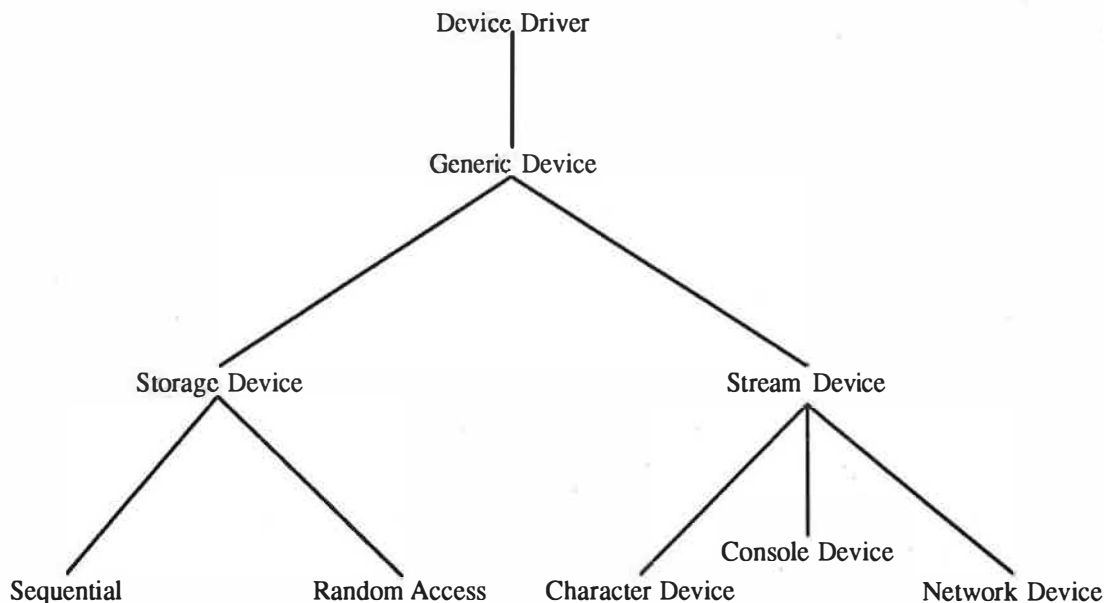


Figure 6: Current Driver Class Hierarchy

them is an object that represents the device driver itself. This object is used to manage the operation of the device driver itself and includes such functions as terminating the device driver. Other objects that are exported represent the devices managed by the device driver. The operations that are typically performed on this type of object include such things as "open". When a device is "opened", a new object is created by the device driver that represents an active or opened instance of the device. It is the active device instance upon which such operations as read and write are performed. All of these objects are represented by ports. There are no strings used to identify the various objects as is the case with `device_open` in current Mach systems. We are able to avoid the use of strings by using the name service to perform the string-to-port translation prior to a client communicating with the device driver.



## 7. Internal Structure

Internally, our device drivers are structured as C threads programs and are dependent only on the microkernel, the HRM, ANSI C library routines and other personality-neutral services. In particular, they do not require that a Unix system personality be present in the machine. Figure 7 shows a typical device driver's internal structure. It has C threads to receive service requests, to handle *Yield* messages from the HRM and to receive interrupt events and multiple C threads to support the execution of the device control logic. Each device is usually managed using a controller state machine that tracks the current state of the device and the possible state transitions it may make. The transport layer hides the low-level details of accessing the hardware to increase the portability of the device driver code. The device driver generally manages a number of queues and exports a number of objects that are "openable". It may also have one or more active instances representing "open" objects.

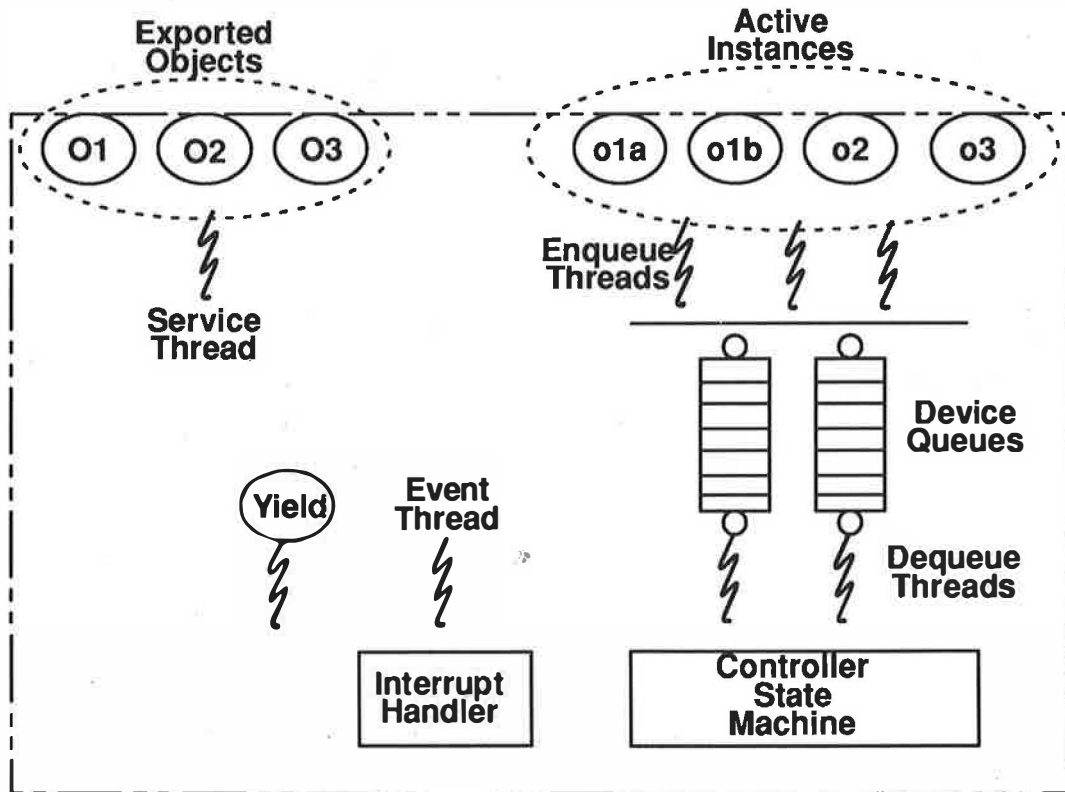


Figure 7: Sample Device Driver Internal Structure

All of the device drivers that we have implemented utilize at least three threads. Often the number of threads that are contained in a device driver is dependent upon the number of "open" objects. The threads are divided into one of four categories. The service thread is the thread that waits for messages that are directed to the objects exported by the device driver. The yield thread responds to *Grant/Yield* requests from the HRM. The event thread is responsible for interactions with the device driver's interrupt handler. The worker threads are responsible for processing messages that are directed to the active objects exported by the device driver. They are also responsible for processing the device queues and running the controller state machine that may exist.

## 8. Microkernel Support

To support the device driver model described above, we have extended the microkernel to provide the mechanisms necessary for efficient access to the various hardware resources. Some of the support is used exclusively by the bus managers responsible for the bus(es) to which the resources are physically attached. Other interfaces are used directly by the device drivers.

## 8.1. I/O Object Services

The I/O object service interfaces are used only by the bus managers to manage directly I/O objects such as memory-mapped I/O and I/O ports. There are two new microkernel objects that allow user tasks to access the I/O objects. The first is the master I/O memory object, which represents a memory object exported by the microkernel that encompasses all of the I/O memory in the system. In some implementations, this memory object represents only a portion of the physical address range. In other implementations, such as the Intel 80386/80486, this memory object represents the entire processor physical address range. Portions of this memory object can be mapped into user tasks through the use of an extension of the external memory manager interfaces called `memory_object_remap_pages`. This causes the actual physical page frame to be supplied to the user task rather than a copy of the page. This is necessary because the semantics of I/O memory are that the physical page frames must be supplied and not just copies. The second new object is the master I/O port object. This is mainly useful only on processors that have a separate I/O address space that are accessed by special I/O instructions such as the Intel 80386/80486. The interfaces that use the master I/O port object mainly relate to the Intel processor architecture. The interfaces allow individual threads to be granted access to specific I/O ports.

## 8.2. Interrupt Handler Services

These interfaces are used only by the bus manager to manage the injection of interrupt handler code into the microkernel's address space. Device drivers can inject small amounts of code into the kernel to be executed there. We have taken this approach rather than those described in [2] because this gives greater flexibility over a wide range of processor and platform architectures than other possible mechanisms. It also permits us to re-drive I/O at interrupt time if desired to increase the efficiency of I/O and to improve device utilization.

The services that were added to the kernel to support interrupt handlers are divided into three groups. The first group is used to load and unload the interrupt handler code into the microkernel. The second group is used to attach and detach the interrupt handler to a specific interrupt level. The final group is used to block and unblock interrupts from being delivered. The HRM is the only direct user of the first two groups. It provides higher level functions that can be used by device drivers.

Interrupt handlers are loaded into the microkernel in three stages. The first stage is to define the code and shared data regions to the microkernel. This also returns the kernel virtual address where the interrupt handler code will be loaded. This allows relocation to be applied to the code if necessary. The kernel does not actually load the code; instead, it is injected into the kernel's address space. The second stage is to inject the code into the microkernel. The final stage is to define the code as an interrupt handler. This final step does two things. The first is to define the various I/O objects that the interrupt handler uses. The second is for the microkernel to pass back a port that can be used as the parameter for the I/O semaphore. Interrupt handlers can be removed from the microkernel either by issuing the message to the kernel specifically for that purpose or by detecting that all of the send rights to the interrupt handler port given by the microkernel have been deallocated.

When an interrupt handler is defined and loaded into the microkernel, it is not called until it is attached to an interrupt level. Interrupt handlers can be attached to multiple interrupt levels if desired by the device driver. Interrupt handlers can also be detached from the interrupt level. Whether an interrupt handler is attached to an interrupt level or not does not affect its load status. These interfaces are used by the HRM to manage which interrupt handlers will receive interrupts depending upon the *Grant/Yield* status of the resource instance that represents the interrupt handler: interrupt handlers are automatically attached and detached as a device driver is granted and yields the hardware resources.

Most device controllers cannot have an I/O operation started in a single CPU instruction. If the controller also supports multiple simultaneous operations, some mechanism must be provided to keep the interrupt-handler from being executed while some low level command is issued to the controller. Most controllers provide for an interrupt enable bit in a status register which prevents the controller from presenting interrupts to the rest of the system. However, there are a small number that do not have this capability. We have, therefore, added a set of interfaces to the microkernel that allow interrupts to be masked and

unmasked at the interrupt controller. This is suitable for MP systems because it does not involve changing a processor's ability to receive interrupts, but rather affects only the interrupt delivery hardware. Systems that do not have a mechanism for manipulating the hardware must employ some alternative internal mechanism to prevent an interrupt handler from being executed.

Interrupt handlers execute in the same state and address space as the rest of the microkernel. While an interrupt handler can access any data structures or code in the microkernel, the only legal access an interrupt handler has to the microkernel are those provided in its parameter list when it is called to process an interrupt. The parameters that are passed to the interrupt handler include the interrupt vector number, a list of I/O ports it can use, a list of I/O memory locations that it can use and a list of microkernel functions that it may call to have services performed on its behalf by the microkernel. The microkernel services include posting an I/O semaphore, starting a DMA operation, aborting a DMA operation, unwiring memory and calculating a kernel virtual address from a physical address.

### 8.3. Interrupt Delivery Services

The kernel provides a simple mechanism for the delivery of events from an interrupt handler to its device driver. It also provides for a shared memory region for the communication of information between the interrupt handler and the device driver. We have adopted a basic protocol that all device drivers and interrupt handlers use to communicate with each other.

The region shared between the device driver and the interrupt handler is used to contain one or more queues used to communicate events from the interrupt handler and the device driver. In some cases the shared region is used to contain requests that the interrupt handler should start when a previous operation completes. This allows a minimum turn around time for I/O re-drive. The interrupt handlers that have been written to date only perform I/O re-drive when the previous operation, the one that cause the interrupt, was successful. This reduces the complexity of the interrupt handler and the time spent on the interrupt level. By convention, the shared memory region has at its start a header that describes the queues that are contained in the region. Each queue is described by a queue header which describes the current state of the queue. Each queue contains a queue header and a collection of queue elements. Figure 8 illustrates a sample shared region that contains an input queue so the interrupt handler can deliver events and information back to the device driver and an output queue that allows the device driver to provide commands to the controller that can be sent by the interrupt handler after the interrupt handler has finished processing an input event.

The queue header describes the contents of the queue. All queues are a fixed number of bytes in length. It is the state of various fields in the queue header and in the queue elements that describe the amount of useful data contained in the queue. The first field in the queue header describes the length of a queue element. All elements in the queues are fixed length. The second field indicates the number of queue elements contained in the queue. The third field contains the number of queue elements that are in use and are in the queue. For example, if the queue is empty, this field contains zero. If the queue is full this field equals the number of queue entries that can be contained in the queue. The fourth field is the queue head. This is an offset from the start of the shared region to the next element to be processed by the consumer—the interrupt handler for output queues and the device driver for input queues. The final field is the queue tail. This is an offset to the last element that was placed on the queue by the producer—the device driver for output queues and the interrupt handler for input queues.

Queue elements contain several fields. The first field is a set of flags that describe the state of the element. The element can be unused, inuse, locked by the producer or locked by the consumer. The next field contains the number of bytes of valid data that are contained in the data portion of the element. This allows variable amounts of information to be contained in a queue element. Finally, there is the data that is contained in the queue element. What the data represents and what should be done with it are determined by mutual understanding of the interrupt handler and the device driver.

Figure 9 illustrates the life cycle of a queue entry on an input queue. This life cycle shows the various states that a queue entry can be in as indicated by the various flags contained in the queue entry. Each

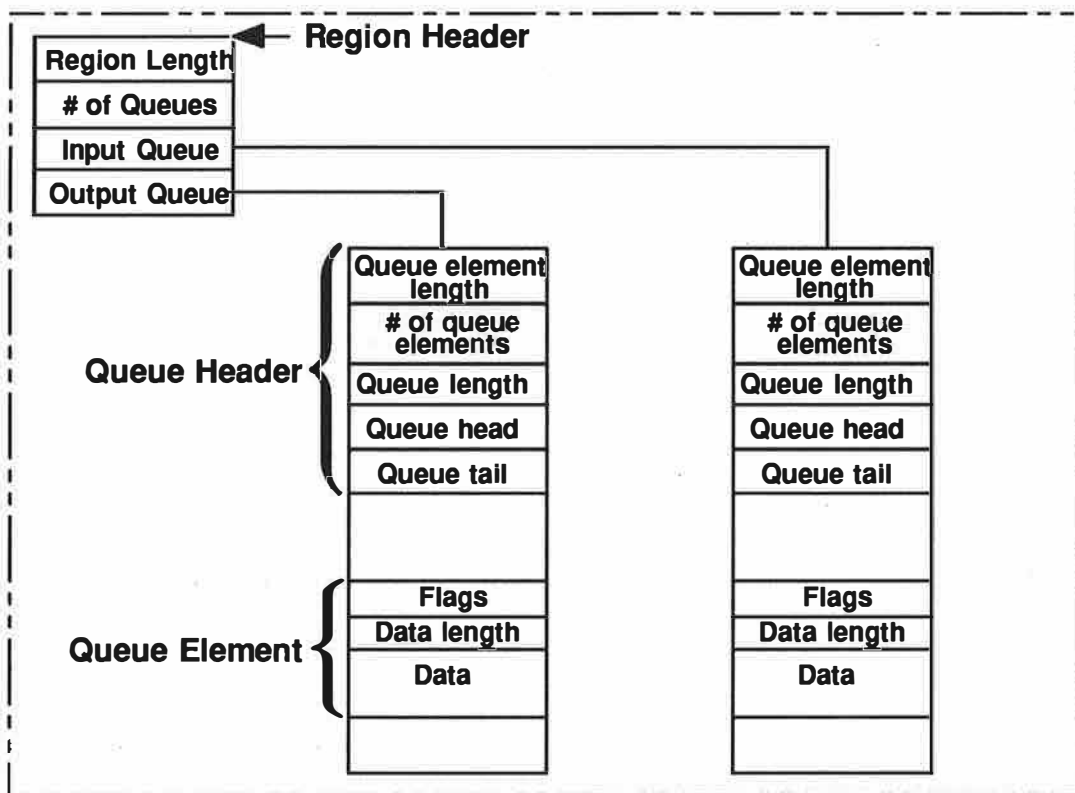


Figure 8: Sample Device Driver/Interrupt Handler  
Shared Region

state is described below. The life cycle of a queue entry on an output queue is similar except that the roles of the interrupt handler and the device driver are reversed.

#### Not Queued

This is the *idle* state of a queue entry. The state of the queue entry is **UnLocked** and **UnUsed**.

#### Locked by Interrupt Handler

This is the state where the interrupt handler first acquires the queue entry. It does this principally by setting the lock state to **ProdLock**.

#### Updated by Interrupt Handler

This is the state in which the interrupt handler puts information into the queue entry. Many of the fields are updated but the principal indicator is that the status is set to **InUse**.

#### On Queue

This is the state in which the queue entry is considered to be on the queue. At this point the lock state of the queue entry is **UnLocked** and the status of the queue entry is that it is **InUse**. Prior to this happening, the interrupt handler updates the queue tail offset.

#### Processed by Device Driver

This is the state in which the queue entry is being processed by the device driver. At this point the lock state of the queue entry is **ConsLock**.

#### Not Queued

This is the final state of a queue entry. The state of the queue entry is **Unlocked** and **UnUsed**. This state is entered by the device driver unlocking the queue entry and marking it to be not

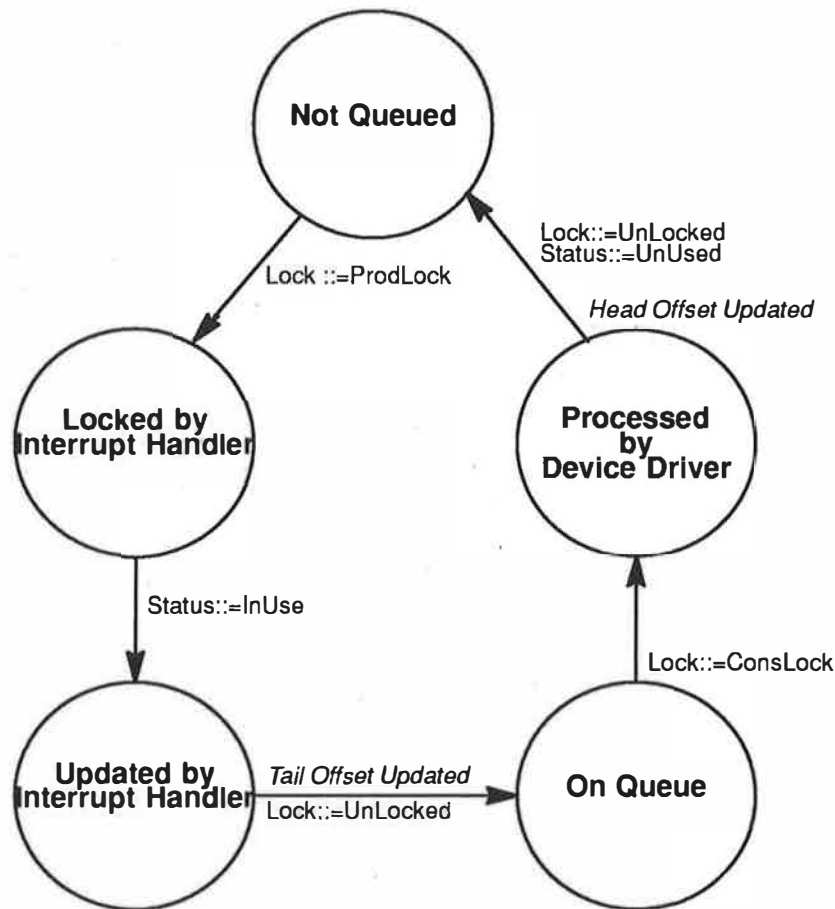


Figure 9: Queue Entry Life Cycle

in use. At this time, the device driver also updates the queue head offset to the next queue entry in the queue.

Synchronization between the device driver and its interrupt handler is accomplished through the use of an I/O semaphore. This allows a user thread to wait for an event to be generated by the interrupt handler. The interrupt handler is capable of posting the semaphore causing the thread to be awakened. Usually the interrupt handler does not post the semaphore each time it is entered. The interrupt handler usually posts the device driver only when the status of the queue changes from empty to non-empty on input queues or full to not-full on output queues. There is an I/O semaphore defined for each interrupt handler. It is represented by the port that is passed back by the microkernel when the interrupt handler was defined.

#### 8.4. DMA Services

The kernel provides a set of DMA primitives that device drivers can use to support devices which perform I/O through DMA operations. We have left the management of DMA in the microkernel for several reasons. First, for first-party DMA the operations required are primarily virtual memory manager manipulations. Second, the DMA hardware is primarily platform- rather than device-dependent, making it reasonable to include it in the microkernel, which is the primary isolating layer for the platform. Finally, we left DMA in the microkernel due to the level of contention for the DMA hardware among the different device drivers in the system.

The support for first-party DMA is based upon I/O wiring lists. These lists represent data structures that are provided by the microkernel to represent memory regions that have been wired by a device driver.

There are interfaces to the microkernel that allow I/O wiring lists to be allocated and deallocated by a device driver. Once a device driver has acquired one or more I/O wiring lists, it can use them to wire memory. The I/O wire interface differs significantly from the `vm_wire` interface in that once the kernel has wired the indicated region, a list of physical addresses is returned. These addresses can be used to program the first party DMA device. There are corresponding interfaces to unwire a wired region and to deallocate an I/O wiring list.

The third-party DMA services export generalized DMA controller functions. The basis for DMA operations is a DMA channel. Device drivers can allocate DMA channels for their use. This causes the microkernel to establish a DMA channel for use by the device driver. Once a DMA channel has been allocated, a device driver can start a DMA operation. The microkernel programs the DMA hardware as appropriate for the parameters to the start operation. There are interfaces to allow a device driver to query the status of a DMA operation and to abort a DMA operation that is in progress. There is also an information call which returns the actual capabilities of the DMA hardware such as the maximum size of a DMA transfer and the ability of the DMA hardware to support transfers spanning multiple physical pages.

## 9. Implementation Experiences, Future Directions and Related Work

We have implemented an HRM for the IBM PS/2 that should be readily adaptable to other similar hardware such as Intel x86 ISA and EISA bus machines. We have also implemented all of the microkernel support described above. We implemented and demonstrated in September, 1992, a user-level console device driver: this driver supports multiple operating system personalities, switching among them and our implementation of X11R5 and Motif®. We are currently writing a complete suite of device drivers for both the PS/2 and x86 ISA bus machines including floppy, ESDI hard disk, SCSI hard disk, token ring and Ethernet. As of this writing in very late February, 1993, the ESDI device driver is functionally complete and is able to read and write disk blocks. Once the initial set of device drivers is completed, we expect to replace all of the in-kernel device drivers that we are currently using and to remove the `device_*` interfaces from the microkernel. We will then begin an intensive performance measurement and tuning effort. Although early experience has shown this device driver model to be viable in terms of its performance, a full test awaits the completion of the complete device driver suite.

Our device driver architecture for Mach 3.0 differs in detail from that of [2], but it represents further development along similar lines. There is one additional architectural problem with Mach 3.0 that the introduction of user-level device drivers and the removal of all in-kernel device drivers makes particularly acute—that of bootstrap. We have solved this problem as well, and we will report on our solution in a separate paper[4].

## 10. Summary

We have undertaken to provide a comprehensive device driver model and implementation that is appropriate for microkernel-based systems that support

- multiple operating system personalities
- real time operation
- modular development and packaging.

This work represents the efforts of a number of individuals at IBM Boca Raton and IBM Austin. We have also benefitted tremendously from the assistance of the Mach and RT Mach projects at Carnegie Mellon University and from the work of the OSF Research Institute.

## REFERENCES

- [1] D. Black, et al., "Microkernel Operating System Architecture and Mach," *USENIX Microkernels and Other Kernel Architectures Workshop Proceedings*, 1992, pages 11–30.
- [2] A. Forin, D. Golub, B. Bershad, "An I/O System for Mach 3.0," *USENIX Mach Symposium Proceedings*, 1991, pages 163–176.

[3] G. Sotomayor, F. Rawson, D. Golub, " An Architecture for User Level Device Drivers for Mach 3.0," to appear in *Proceedings of the Third USENIX Mach Symposium*, 1993.

[4] G. Sotomayor, F. Rawson, " A Bootstrap Architecture for Mach 3.0" , forthcoming.

UNIX is a registered trademark of UNIX System Laboratories, Inc. Motif is a registered trademark of the Open Software Foundation. AIX, IBM and PS/2 are registered trademarks of the IBM Corporation.





# MVM – An Environment for Running Multiple Dos, Windows and DPMI Programs on the Microkernel

David B. Golub

*School of Computer Science  
Carnegie Mellon University  
Pittsburgh, Pennsylvania 15213*

Ravi Manikundalam

*International Business Machines Corporation  
Austin, Texas 78758*

Freeman L. Rawson III

*International Business Machines Corporation  
Boca Raton, Florida 33431*

## Abstract

The most significant feature of the personal computer industry today is its extremely large body of legacy software that is built on the original PC Dos system and the initial implementation of BIOS. This software has grown and been extended by Dos-based system extenders such as Microsoft Windows®. Many of these environments go beyond the original real mode offered by the original PC and use the 16- and 32-bit protected modes of the Intel® 286 and 386 parts. In order to be able to run PC legacy software on our Mach-based systems, we have designed and implemented a virtual machine environment known as Multiple Virtual Machines (MVM) for emulating Dos, Windows and the Dos Protected Mode Interface, an industry standard for coordinating and supporting protected mode Dos extenders. We provide support for multiple virtual machines executing on the microkernel, capable of running both DOS real mode and extended DOS protected mode programs. This support allows multiple applications to share access to all of the features of the underlying hardware, including low level access to the native devices and virtualized extensions to device services. All the principal features of this system have been implemented and demonstrated both stand-alone and in conjunction with other operating system personalities such as Unix® and OS/2™. We have successfully run many DOS, protected mode extended DOS and Windows 3.x applications on our stem. In order to do this, we not only used the existing facilities of Mach to implement our client/server emulation cradle, but we also extended the microkernel to support certain features that we needed to achieve the right levels of performance and function.

## 1. Introduction

There have been several efforts to design and build emulation systems based on the Mach microkernel to emulate various operating system personalities ranging from DOS to VMS[1,2,3,4,8]. The approach taken by each of these efforts has ranged from emulating the native operating system with the help of a personality-specific centralized server[1] to having a collection of personality-neutral servers that provide services rich enough to emulate a range of operating system personalities[5]. There is a large effort currently underway within IBM to implement a general purpose architecture for emulating multiple operating system environments including Unix, OS/2, DOS, and Windows, concurrently on the microkernel. Several groups within IBM have been working on this project, designing and building proof of concept models in such areas as user-mode device drivers[11], file

servers, and personality—specific emulation support for non—Unix personalities such as OS/2[6]. These prototypes are being used to confirm the feasibility of using the Mach microkernel as a general purpose platform for running multiple operating system personalities concurrently and to assist in identifying extensions and changes that are needed to the microkernel to support systems that do this. Our project is a part of this larger activity.

Our goal is to design a virtual x86 machine environment on the microkernel that provides the same high quality, seamless support for DOS, protected mode DOS (DPMI) and Windows 3.x applications that is delivered by OS/2 2.x. At the same time, we wanted to build a prototype that would help answer some questions about the feasibility of using Mach to support such an environment. We were encouraged by the success of the work done at Carnegie Mellon[4] in implementing support for the i386/i486 family's "virtual 8086" mode that provides a hardware environment for running real mode programs without disabling the protected mode's protection and paging features and the real mode Dos and BIOS environment. But we needed to ensure that our multiple virtual x86 machine environment on Mach that would not only support real mode programs but also the protected mode features of the 286 and 386 families of processors.

Our implementation includes:

- a Multiple Virtual Machine (MVM) server that exports a message—based API to create, configure and destroy virtual machines
- a collection of Virtual Device Drivers (VDDs) that virtualize or emulate different aspects of the PC and x86 environment.
- a collection of machine—dependent and machine— independent kernel support for virtual 8086 mode and protected mode support in a virtual machine.

Our VDD model is open allowing users to implement new VDDs to extend the default device and emulation support in a virtual machine. The use of VDDs permit applications running in a Virtual Machine to share devices with applications of other operating system personalities like Unix or OS/2 that may co—exist in the system.

This paper starts with a brief overview of the environments that need to be supported in a virtual machine. This is followed by a description of our design, implementation and experiences, a discussion of the extensions that were made to the microkernel to support our work. We conclude with a review of our current implementation's completeness and performance as well as a brief mention of some possible extensions.

## 2. An Overview of the PC Dos Environment

In order to understand the challenge of supporting the x86 software legacy on Mach, we first have to understand something of the x86 hardware and how it evolved from the 8086. The key features of the 386 and following parts are:

- **Real Mode.** In this mode, the 386 processor is essentially an 8086.
- **Protected Mode.** In protected mode, the processor supports 16—bit segmented, 32—bit segmented and 32—bit flat programming models. It also provides paged address translation underneath the segmented architecture.
- **Virtual 8086 Mode.** The processor provides a virtual set of real mode registers and directly interprets memory and executes instructions as if it were running in real mode. Certain instructions that cause the interrupt state of the machine to be altered or otherwise might interfere with protected mode operation are considered "privileged" in this mode and cause exceptions to be raised when encountered.

- **IO Bitmap.** The 386 architecture provides mechanisms to protect I/O from user mode programs (protect mode or v8086 mode) on a selective basis using the IO Permission Bit Map (IOPM).

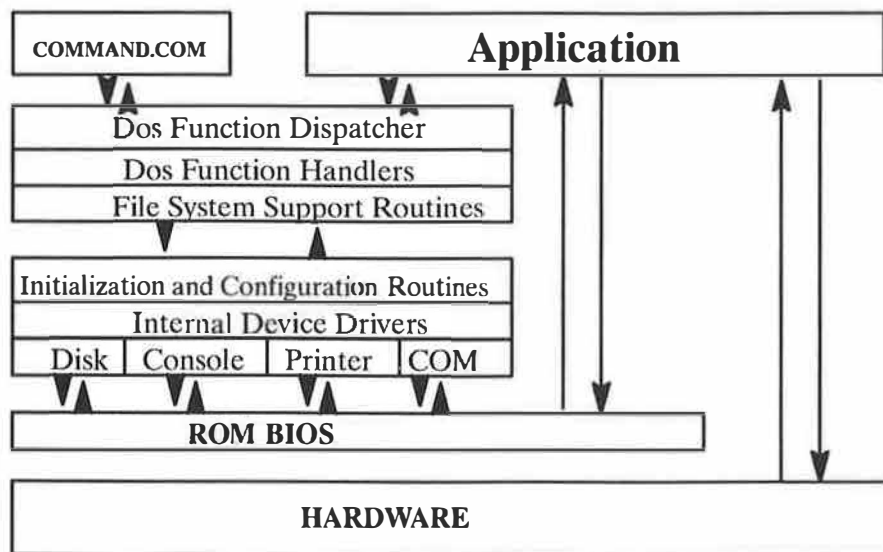
The basic structure of Dos and the flow in a Dos application can be pictured as shown in **fig 1**. There are four different layers, each of which provide different levels of services to an application or user in the Dos environment.

- COMMAND.COM, the Dos command line shell
- Dos kernel which includes the file system, character and block devices and some simple program management
- BIOS which provides a primitive set of hardware – specific device and system interfaces
- the hardware itself.

There is nothing that prevents a Dos application from using any of the services that are available at any of the levels all the way down to accessing the raw hardware. Dos executes in the real mode of the Intel x86 family of processors as a single user, single tasking environment.

Dos applications make service requests by loading the parameters for the call into machine registers and executing a software INT NN instruction. Dos applications typically use some combination of the following as shown in **fig 1**.

- Dos services via INT NN (NN = 20h, 21h, 24h, 25h and 26h)
- BIOS services via INT NN (NN=10h, 13h, 15h etc)
- program directly to hardware
- shut on and off interrupts in the machine at will



**fig 1: Structure and flow in a Dos environment**

Since Dos and Dos programs execute in real mode, there are some inherent limitations including

- no multitasking
- a 1 megabyte limitation on addressability. In fact since the upper portions of the 1 megabyte are used to map IO memory, there is only 640KB available in the Dos environment.

In response to these limitations a number of Dos extenders developed over the years including Microsoft Windows. Since the 386 architecture does not provide any hardware support for full virtualization of 286 or 386 protected mode programs, extended DOS programs such as Windows 3.x need well-defined protocols to coexist. The Dos Protected Mode Interface (DPMI)[14] provides a controlled mechanism that lets multiple extenders and extended applications work together. The DPMI specification defines functions to

- request and manipulate protected mode resources like local descriptor tables, extended memory (memory above 1 Meg), certain CPU control registers, interrupt descriptor tables and page tables.
- allocate and free Dos memory (memory below 1 Meg)
- communicate with real mode programs
- switch execution modes between virtual 8086 mode and protected mode.

In this environment a Dos extender programs like Windows 3.x is referred to as a *DPMI client* and the term *DPMI host* refers to the entity that provides the DPMI services. The MVM environment supports multiple protected mode virtual machines by implementing the DPMI specification.

### 3. Structure of MVM

MVM supports executing multiple Dos, extended Dos and Windows 3.x programs on the Mach microkernel environment. Each Dos or extended Dos program runs in its own x86 virtual Machine. The Windows environment is treated as a single extended Dos program and runs in a single virtual machine. MVM uses the full facilities of the x86 hardware including the virtual 8086 mode and supports the creation and concurrent operation of multiple virtual machines. It is comprised of the following four major components.

- A Multiple Virtual Machine (MVM) server that exports a message-based API to create, configure and destroy virtual machines. The server maintains a database of global and per virtual machine configuration and tuning properties.
- A collection of Virtual Device Drivers (VDDs) that virtualize or emulate different aspects of the PC and x86 environment for the virtual machine.
- A Virtual Machine Monitor (VMM) that services all exceptions in a virtual machine, decodes the instruction that caused the exception and dispatches an appropriate Virtual Device Driver to emulate or virtualize it.
- A module that emulates the Intel 8259 Programmable Interrupt Controller (PIC) and provides a clean and efficient mechanism to simulate external events such as interrupts to a virtual machine.

The MVM environment can be best understood by examining its memory and execution models.

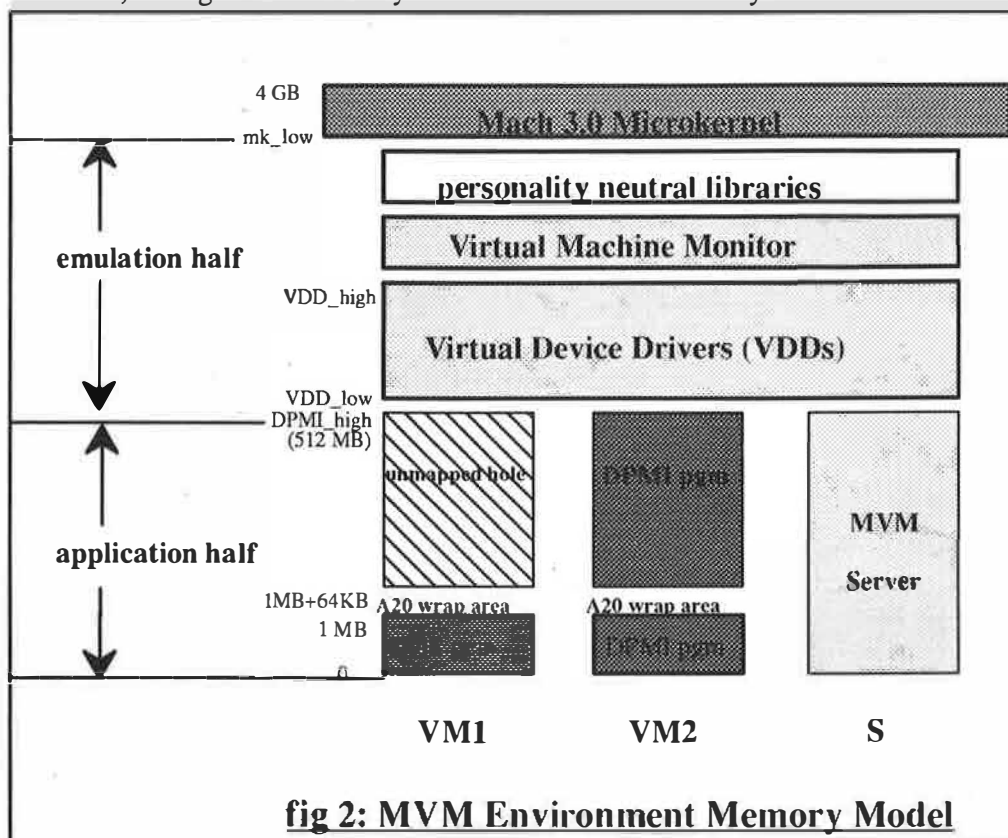
#### 3.1. Memory Layout

In MVM the MVM server and each virtual machine run as separate Mach tasks. Each virtual machine runs either a Dos program in virtual 8086 mode or an extended Dos program as a DPMI client in protected mode. The MVM environment itself plays the role of the DPMI host.

The address space of a virtual machine depends on the type of program being executed in it. **Fig 2** shows the address space of the MVM server, a virtual machine of each type and the relationship between them. **VM1** is a virtual 8086 mode virtual machine address space while **VM2** shows a DPMI client running. **DPMI\_high** sets the upper limit of memory that can be used by the DPMI clients: the DPMI specification defines that such a limit exists, may well be less than 4 GB and provides an interface that permits programs to retrieve this value. Our value of 512MB for **DPMI\_high** is the same as that used in OS/2 2.x. All of the memory between 0 and **DPMI\_high** is reserved for application programs. Above **DPMI\_high** each virtual machine is a clone of the MVM server's address space. Passive master copies of the Virtual Machine Monitor and all the Virtual Device Drivers are loaded and initialized in the MVM server address space and are inherited by every virtual machine that the server creates with **task\_create()** by marking the memory as inheritable.

### 3.2. Execution Environment

The MVM server is a multi-threaded Mach task. It has three threads -- an init/service thread, a notify thread and an exception thread. The init/service thread is the first thread that starts in the MVM server. All the initialization functions including loading the VDDs and VMM are done by this thread. This thread also makes MVM known to other servers in the system by checking its service port into the system name space. After completing the initialization, this thread becomes the service thread in the server, servicing all requests to create, configure and destroy virtual machines. The notify thread receives on a notify



port set and services any **PORT\_DEATH** notifications that may arise because of an abnormal termination of a virtual machine. This lets the MVM server perform any cleanup needed in the event of an abnormal termination of a virtual machine. The exception thread is the exception server for all exceptions not otherwise caught in the MVM environment. The exception thread in the MVM server is primarily a debugging aid.

Each virtual machine is also a multi-threaded Mach task. The target thread executes the Dos, Windows or DPMI program. All code in the application half of the address space runs on this thread. There is also at least one monitor thread to service all of the exceptions generated on the target thread. In addition, there are other I/O and control threads that execute in the virtual machine to run the VDD code. All of the threads in the virtual machine except the target thread are ordinary Mach threads running in a 32-bit, flat address space and are implemented using C threads. The target thread may run in

- virtual 8086 mode
- 16-bit segmented protected mode
- 32-bit segmented protected mode
- 32-bit flat protected mode

or in some combination of these. When executing in virtual 8086 mode, the target thread is running a standard Dos program that may do a number of operations that are potentially in conflict with the environment that Mach as a portable, machine-independent, multiprocessor operating system creates such as:

- perform IO to devices directly
- request Dos or BIOS services with the INT NN interface
- turn interrupts off and on with CLI and STI instructions.

When executing in protected mode, the target thread may do direct I/O and try to manage the interrupt mask itself, but it also typically uses the DPMI interfaces and expects external interrupts to be delivered in accordance with the DPMI specification. Under DPMI an interrupt gets delivered to the protect mode handler for it if one exists and then passed on to its real or Dos mode counterpart by switching the mode of execution of the target thread if necessary before delivering the external interrupt.

The Mach extensions required to support Dos and DPMI that we added to the microkernel are described later. These extensions together with the x86 hardware are used to force the target thread to take a general protection fault exception whenever it tries to execute an instruction that needs to be emulated or virtualized in order to maintain the integrity of the rest of the system. A high level flow of control in a virtual machine is shown in **fig 3**.

#### **4. Virtual Device Drivers**

Dos applications tend to access hardware resources like devices directly. In order for multiple Dos applications, each running in a virtual machine, to access physical hardware devices, each virtual machine must be provided with a set of virtual interfaces to these devices, so that the actions of one application running in a virtual machine do not affect the state of the device as perceived by other entities in the system.

We have defined a general VDD model that supports installable virtualization, allowing the level of device and emulation support available in a Virtual Machine environment to be extended dynamically without requiring upgrades to the rest of the system. VDDs can be classified into four categories.

- step 1:** target thread executes a privileged instruction and causes an exception and traps to the kernel
- step 2:** Kernel sends exception message to the Virtual Machine Monitor
- step 3:** The Virtual Machine Monitor gets the state of the target thread
- step 4:** Virtual Machine Monitor decodes the instruction that caused the exception
- step 5:** if instruction needs to be virtualized  
           else call appropriate VDD to virtualize it  
                   let the event be handled natively
- step 6:** set the state of the target thread
- step 7:** resume the target thread by replying to the exception message

**fig 3**

- VDDs that arbitrate for ownership of a device or resource on behalf of the program or native Dos device driver in order to allow direct access to the device or resource. Examples include floppy, communications ports, DMA channels, display and devices not supported by the rest by the system for which a Dos device driver exists.
- VDDs that virtualize a device or emulate a service by communicating with other servers in the system like device drivers or a file server. Examples include keyboard, mouse, file system services and the hard disk.
- VDDs that emulate various Dos memory extensions like EMM [12], XMS [13] and DPMI itself.
- VDDs that virtualize various other hardware aspects of a PC environment such as the timer, the interrupt controller and the numeric coprocessor.

To simplify the coding of the VDDs, we have taken from OS/2 2.x the notion of a library of helper routines called Virtual Device Driver Helper (VDH) services that make it easier to plug a VDD into the MVM environment. These services provide a VDD with mechanisms to

- install event handlers. This allows a VDD to be notified upon events such as virtual machine creation and virtual machine termination so that the VDD can perform any per virtual machine initialization or cleanup.
- install virtualization or emulation handlers. This allows a VDD to be notified of events that it is supposed to handle.
- provide access to a virtual interface to a programmable interrupt controller (PIC). This provides a standard mechanism that the VDD can use if it needs to simulate external events such as interrupts to a virtual machine.
- register and query values from the per virtual machine configuration data base that is maintained by the MVM server. This allows a VDD's behavior to be configured or tuned on a per virtual machine basis.
- make upcalls into the application that is executing in the virtual machine. A good example of the use of this is to invoke Dos or real mode services on behalf of the protect mode application that is running in the virtual machine.
- allocate and arm return hooks. This provides a VDD with a mechanism to regain control after the target thread is done handling an upcall or to set up an event handler for external events such as interrupts.

The return hooks are based on the notion of virtual 8086 and protected mode breakpoints: these breakpoints are illegal instruction sequences that are specially recognized by the VMM and permit MVM to set in the target program places where it gets control to emulate or virtualize the PC hardware environment. These mechanisms are needed, for example, to provide support for Dos Terminate and Stay Resident(TSR) programs and to perform a more complete virtualization of the keyboard and mouse interrupts.

Some of the work that is reported in [11] develops a hardware resource arbitration protocol and a set of services that allow VDDs to arbitrate and gain direct access to hardware resources. Indeed, the whole hardware-sharing protocol was originally motivated by the problems of sharing devices with and among virtual machines. See [11] for the details of this protocol and our user-level device driver model.

#### 4.1. Structure of Virtual Device Driver

A virtual device driver or VDD is an installable module in the MVM environment. It typically has the following :

- **VDD initialization handler.** This does any global initialization of the VDD such as installing virtual machine creation and termination handlers, registering any configuration or tuning parameters that it may support and allocating any global resources that it may need.
- **Virtual machine creation handler.** This does any per virtual machine VDD initialization like installing exception or virtualization handlers, establishing connections with other entities in the system such as the file server or a device driver and allocating any per virtual machine resources that it may need.
- **Virtual machine termination handler.** This is responsible for any cleanup that may have to be done at virtual machine termination such as freeing certain resources and closing connections with devices.
- **A list of emulation and/or device virtualization handlers.** These get dispatched by the Virtual Machine Monitor depending on the instruction or event that it determines needs to be virtualized in the Virtual Machine.
- **Grant\_Yield handler.** This is optional. Typically it exists only in VDDs that need to arbitrate for the ownership of a device and grant direct access of the device to the program executing in the virtual machine. The Grant\_Yield handler is usually dispatched by the Hardware Resource Manager(HRM) [11].
- **Interrupt handler.** This is also optional. Typically it exists only in VDDs that obtain the ownership of an interrupting device from the HRM and that must then handle interrupts from the device.
- **VDD termination handler.** This is responsible for any global cleanup of resources that may need to be done before VDD termination.

#### 4.2. Execution Environment

The VDD initialization and termination handlers execute in the context of the MVM server and most of the rest of the VDD executes completely in the context of the virtual machines. In general the part of the VDD that emulates a certain Dos, DPMI or BIOS service invoked with the INT NN interface executes on the virtual machine monitor thread in the virtual machine while the part of the VDD that virtualizes a device or an external event executes on its own thread in the virtual machine. If the VDD has a Grant\_Yield handler, it too has its own thread in each virtual machine. It is the responsibility of the



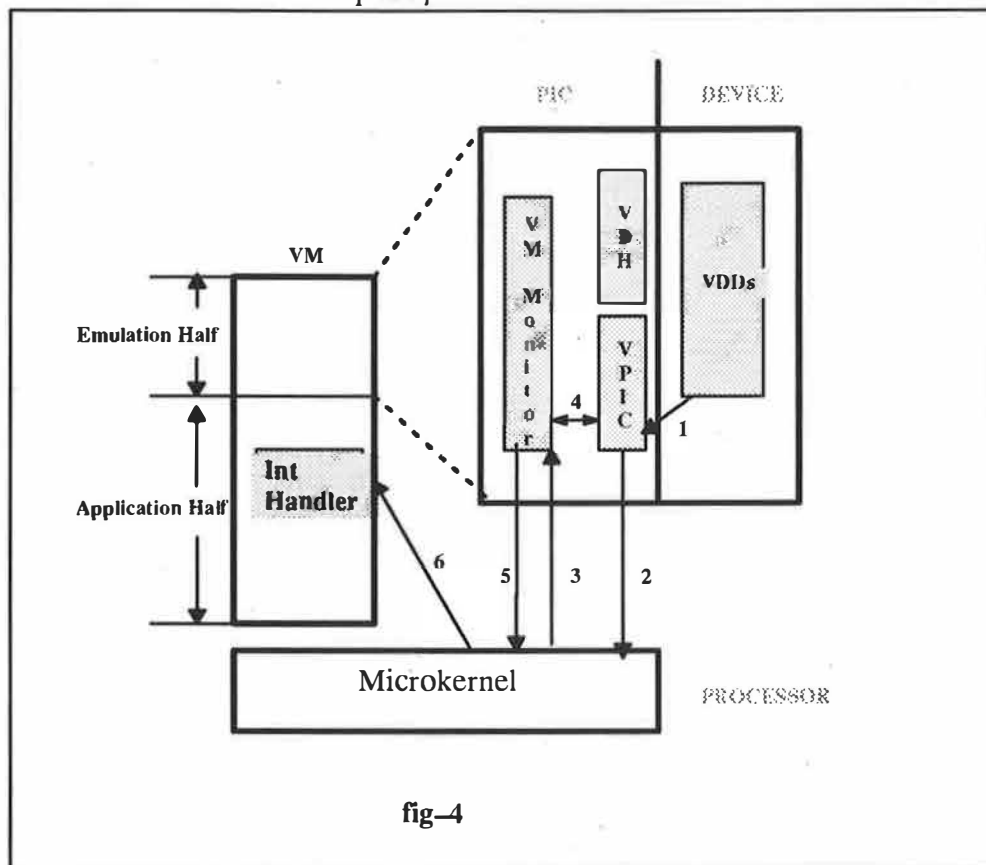
Grant\_Yield handler to save or restore the state of the hardware or device on behalf of the program that executes in the virtual machine. The interrupt handler gets invoked by the microkernel [11] to deliver an interrupt from the device to which this VDD has gained access. The interrupt handler executes on its own thread in the VDD. VDDs that need to simulate external events to the virtual machine do so by using the standard set of virtual interrupt services that are provided as part of the VDH interfaces.

#### 4.3. VDD Loading and Unloading

Some VDDs are essential to MVM operation: these are referred to as *base VDDs*. However, there may be other VDDs that extend the MVM environment: these are called *optional VDDs*. Base VDDs get loaded during boot/initialization time by the MVM server. They are present as long as the MVM environment is running on the system while any optional ones can be loaded into and unloaded from MVM upon request.

#### 5. Interrupt Delivery

Fig -4 describes the flow of control during interrupt delivery to a virtual machine in the MVM environment. This flow is similar to the one used to deliver interrupts in the native PC environment. Interrupt delivery begins when the VDD raises a virtual IRQ line by using the Virtual PIC services that are provided by the MVM environment. This is very similar to the device raising the interrupt on PC hardware. The software module (VPIC) that is emulating the PIC now determines if an interrupt at this IRQ level can be accepted by looking at its virtual Interrupt Request Register (IRR) and virtual Interrupt Mask Register (IMR) values: these are virtualizations of the corresponding registers in the 8259 interrupt controller. If the VPIC determines that the virtual machine can service this interrupt now, it now makes a software interrupt request to the microkernel. This is similar to the



PIC raising the interrupt line to the processor. The microkernel checks interrupt state of the target thread in the virtual machine to which the interrupt needs to be delivered. If it is interruptible, the microkernel requests the VMM in the virtual machine to provide it with the address of the interrupt vector in that the virtual machine to which it should vector. This is very similar to the processor driving the interrupt acknowledge line in the PC hardware. The VMM consults the VPIC to get the interrupt vector and supplies this information to the microkernel. The microkernel sets the state of the target thread to start executing the interrupt handler when resumed and resumes the thread.

Since we emulate the 8259 PIC completely in software and use new support in the kernel for delivering software interrupts to a thread, we have a clean mechanism to simulate or deliver interrupts to the target thread in the virtual machine without explicitly suspending and resuming the target thread. This also provides us with a mechanism to support the interrupt delivery semantics required by the DPMI specification without adding undue complexity to Mach.

One of the goals of the MVM environment is to support native Dos device drivers for character mode single-user devices in the system. In order to do this we have a stub VDD that negotiates with the HRM to get exclusive access to the resources that are needed by the device driver. It also registers a `Grant_Yield` handler with the HRM. Once the resource is granted, it makes use of the HRM to register a stub interrupt handler for the device with the interrupt manager in the microkernel. MVM then gives direct access to the device to the DOS program or device driver executing in the virtual machine. The Dos device driver does direct I/O to the device. Interrupts from the device are reported to the stub VDD interrupt handler that was registered with the interrupt manager in the microkernel. This handler then delivers the interrupt to the actual interrupt handler in the Dos device driver by using the interrupt delivery scheme described above.

## 6. Windowing

In order to support execution of virtual machines in a windowed environment like X11, we need to virtualize all access to the display in a virtual machine and map it to the pixels of an X11 window. Unfortunately, there are many different ways in which the display hardware is accessed in the DOS environment:

- video BIOS (INT 10H services)
- access to the display hardware IO ports
- direct access to the video RAM (VRAM) in text, bit plane or packed pixel mode.[9]

We have been experimenting with various schemes to emulate or virtualize all of these methods of using the display hardware.

We support full video BIOS level emulation by reimplementing the VGA BIOS services [10] as part of our video virtualization support. These services operate on a virtual VRAM structure and a set of virtual hardware registers instead of the physical hardware resources. This lets us introduce some efficiency enhancements since we trap hardware activity at a high level.

The virtual display memory modifications made through the BIOS emulation support mark all of the changes in a bitmap. At the expiration of a timer interval, all virtual display memory modifications to that point are processed in a batch, and the modification bitmap is cleared. The code processes the modifications by generating a display list, a linked list

that describes the regions of the virtual display memory that have been modified since last update. The display list is traversed by the display memory translation routines. The appropriate portions of memory are transformed into an X image which then is displayed in the associated X window.

Virtualizing or emulating direct access to the VRAM from a virtual machine is a hard problem. The manner in which regular or conventional memory is organized is different than the manner in which VRAM is organized, so that there is no straightforward way to use a region of regular memory as virtual VRAM and then make it accessible to the virtual machine by mapping it into appropriate range of addresses that is interpreted by the Dos, Windows or DPMI as the VRAM region. An obvious alternative is one in which every access to the range of addresses that should be occupied by the VRAM in the virtual machine causes an invalid memory access exception. Then an exception server thread can receive the exception message that the microkernel generates and emulate the VRAM access operation in software. However, due to the high rate of accesses to video memory is accessed in a virtual machine (at least one access per pixel draw), this approach has too much overhead to provide us with acceptable performance for graphical applications running in virtual machines. To solve this problem we are currently investigating other approaches to virtualizing the video hardware. One possible approach is to divide excess VRAM, VRAM that is not being used by X and the native graphics libraries, into page granular chunks and use that as a demand paged pool of virtual VRAM. The major open question is whether there is always sufficient excess VRAM to make such a scheme viable.

## 7. Microkernel Enhancements

When we started this project, we found that as part of the Carnegie Mellon work on running Dos on Mach[4], there had been several enhancements in the 386 machine-dependent code of the microkernel to support running Mach threads in virtual 8086 mode. As had already been recognized by the group at CMU, although these changes had been sufficient to get Dos itself and real mode Dos programs running on Mach, additional changes are needed to run the full Dos and extended Dos environment, conforming to the DPMI specification and reaching the levels of function and performance of commercial systems like OS/2 2.x that also provide Intel legacy software compatibility. We also needed to be very careful to ensure that MVM would function correctly in the presence of other, perhaps more than one other, operating system personalities. Since one of us (Golub) had been intimately involved in the CMU activities, we were able to define the needed extensions in a way that was consistent with what had gone on before.

To better describe the changes that we made to the microkernel, we need some terminology.

- A thread that executes in virtual 8086 mode in a virtual machine is called a *v8086 mode thread*.
- A thread that executes a protected mode extended Dos program is known as a *virtual protected mode thread*.

The target thread in a virtual machine is either a v8086 mode thread or a virtual protected mode thread to the microkernel, depending on the program that is currently running in the virtual machine.

Our microkernel changes are in the following areas.

- fast exception handling

- fast reflection of non-emulated exceptions back to the v8086 or virtual protected mode thread in the target virtual machine
- support for interrupt simulation to both v8086 mode and virtual protected mode threads
- facilities that let user-level code manipulate the protected mode resources of the x86 architecture
- support for user-level LDT descriptor management as required by the DPMI specification
- virtual memory features that let a task reserve a portion of its address space for future use

### 7.1. Fast Exception Handling

In the MVM environment the Mach exception mechanism is used to invoke the emulation support in a virtual machine when an instruction in the target thread needs to be virtualized. The Virtual Machine Monitor acts as an exception server in each virtual machine, decoding these exceptions and dispatching the appropriate VDD handler to emulate or virtualize the instructions that caused them. Typically the portion of the VMM that monitors for events that need to be virtualized will be executing the loop as shown in **fig 5**. This makes the exception mechanism critical to the performance of MVM and one of the most important things to optimize.

With the pre-existing exception mechanism in Mach, once the exception server receives the exception message from the microkernel, it has to make two Mach kernel IPC calls — one to get the target thread's state and one to set it. To reduce this overhead we have created some new, overloaded versions of the Mach exception interfaces that combine these calls into the acts of receiving and replying to the exception message. It turns out that the microkernel already has the state saved when it generates the exception message and restores it on the reply if the thread that took the exception is to run again, so there is very little additional overhead or even new code in the microkernel for these interfaces. Our new interfaces include:

- **catch\_exception\_raise\_with\_state** This is identical with **catch\_exception\_raise** except the microkernel supplies the state of the target thread on an exception and accepts and sets the state of the target thread on the return.
- **{task,thread}\_{get,set}\_exception\_handler** This primitive informs the microkernel of the flavor of the thread state that needs to be supplied on **catch\_exception\_raise\_with\_state**.

### 7.2. Fast Reflection of Non-emulated Exceptions

<p><b>step 1:</b> receive an exception message</p> <p><b>step 2:</b> decode instruction that caused exception</p> <p><b>step 3:</b> if instruction needs to be virtualized                    else call appropriate VDD to virtualize it                            let the event be handled natively</p> <p><b>step 4:</b> set the state of the target thread</p> <p><b>step 5:</b> reply to the exception message</p>
---

**fig 5**

As mentioned earlier, all Dos, BIOS and DPMI services are requested when the application loads a set of hardware registers and executes an INT NN instruction. The value of NN determines if it is a BIOS service, a Dos service or a DPMI service being requested. Not all of these services require virtualization as some affect only the virtual machine itself and do not access resources that are being shared with other programs in the system. The 386 architecture specifies the INT NN instruction to be a privileged instruction when operating in the virtual 8086 mode. This results in the situation described in **fig 6**.

We have added support in the kernel for fast reflection of non-emulated exceptions to v8086 or virtual protected mode. This is done by associating what is called an interrupt reflection bitmap as part of a virtual 8086 mode or virtual protected mode thread's processor control block. This bit map is manipulated using a new **i386\_INTERRUPT\_BITMAP** flavor that has been added to the **thread\_get\_state** and **thread\_set\_state** primitives. With this scheme, the kernel actually does the decoding of the instruction that caused the protection fault, consults the thread's interrupt bit map and determines if this is service needs to be emulated or not. If the service does not need to be emulated, then the kernel itself passes control directly to the interrupt handler running on the target thread in the virtual machine. With this scheme the overhead of **step 3** to **step 5** as seen in **fig 3** is avoided for non-emulated exceptions to native mode.

### 7.3. Interrupt Simulation

In the work reported in [4], the CMU investigators had added support for interrupt simulation within a virtual 8086 mode virtual machine by associating a virtual 8086 support data structure called **v86\_assist** that contains a pointer an interrupt queue for the v8086 mode thread and maintains a virtual interrupt flag for it. They had the kernel determine the interruptability of the virtual machine as well as deliver the interrupt to the virtual machine. This scheme has a number of drawbacks.

- The kernel was managing only a virtualization of the processor's interrupt flag and not the state of the interrupt controller as well. Thus, it could not really determine whether a particular interrupt would be delivered at a particular time on a real PC. On PC hardware it is a combination of the state of the processor interrupt flag and the state of the 8259 Programmable Interrupt Controller (PIC) that determines when an interrupt is serviced.
- The kernel could only simulate interrupts during changes to the virtual machine's interrupt state and could not handle nested interrupts at the same level.

**step 1:** Application executes a privileged instruction that need not be virtualized

**step 2:** General protect fault

**step 3:** Exception message to the protect mode monitor

**step 4:** Monitor determines that instruction need not be virtualized

**step 5:** Reflects control back to the native handler

**fig 6**

- The scheme was limited to supporting only virtual 8086 mode interrupts to the virtual machine and did not provide the protected mode interrupt delivery semantics required by the DPMI specification.

We have implemented new support in the kernel for both v8086 mode and virtual protected mode interrupt simulation in a virtual machine. In our implementation we have overcome the above limitations by having the kernel emulate precisely the behavior of the processor interrupt flag. We have provided a full emulation of the PIC in user-level VDD that runs in the virtual machine. In our scheme whenever the virtual PIC (VPIC) VDD determines that there is an interrupt to be delivered, it informs the kernel using a new **thread\_set\_state** flavor called **i386\_INTERRUPT\_REQUEST**. If the virtual machine is interruptible or when it becomes interruptible based on the state of the virtual interrupt flag that the microkernel maintains, the microkernel raises an exception in the virtual machine of type **EXC\_i386\_INTERRUPT**. The VMM acting as the exception server for the target thread determines the interrupt vector of the highest priority interrupt that is pending in the virtual machine by calling a routine exported by the VPIC code and delivers it by setting the correct state when replying to the exception message.

#### 7.4. Access to Protected Mode Resources

When the target thread of a virtual machine executes in virtual protected mode, it

- executes in the segmented model, potentially using directly the segment registers of the 386 family of processors.
- may use DPMI-provided services to
  - ☐ get coprocessor status
  - ☐ set coprocessor emulation
  - ☐ get debug register state
  - ☐ set debug register state.

The intrinsic Mach user-level programming model, being machine-independent does not normally export these features at user level. We have added support that permits virtual protected mode threads to use them as a part of the machine-dependent code in our port of Mach to the 386 architecture. We changed the function of **i386\_THREAD\_STATE** to set the values of the segmentation registers in the cases where the thread is running in virtual 8086 mode or has a separate Local Descriptor Table (LDT) attached to it indicating that it is running in virtual protected mode and has a segmented address space different from the standard Mach address space. This concept had already been under development at CMU by one of us (Golub) and was introduced into our sources on his suggestion. For a virtual protected mode thread that requests a segmented address space, the microkernel creates a separate LDT for the thread and sets the hardware LDT pointer register to point this LDT whenever the thread is dispatched. All of the other threads in the system use a single LDT constructed by the microkernel to create the flat, unsegmented address space required by the standard Mach machine-independent programming model.

In addition, we have added another flavor, **i386\_DEBUG\_STATE**, to **thread\_{get,set}\_state** to allow user-mode access to the hardware debug registers on the 386 family of processors. The microkernel maintains the state of these registers when they are in use on a per thread basis and saves and restores them across context switches when necessary.

The DPMI specification allows a user process to trap the coprocessor (ESC) instructions even if the hardware has a coprocessor: DPMI does this since some existing extended Dos

programs need to supply their own floating point emulation routines. To support this feature we added the `i386_COPROC_TRAP_STATE` flavor to `thread_{get,set}_state`. This flavor lets us turn on or off trapping of floating point ESC instructions. When the microkernel traps them, they are then reflected back through the MVM environment to the application—provided emulator rather than being processed by the hardware or the floating point emulator that is built into Mach itself.

### 7.5. User-level LDT Management

The DPMI specification requires a set of LDT descriptor management services. These have been implemented by using the support for optional per thread LDTs discussed above and adding two new microkernel functions `i386_get_ldt` and `i386_set_ldt`. These functions permit MVM to provide the required LDT access and manipulation functions to DPMI programs.

Since some virtual protected mode threads may have access to their own LDTs, there are protection problems that have to be handled carefully. In the 386 architecture the descriptors stored in the LDT not only describe the use of memory such as code, data or stack but also provide privilege and protection information. One way to switch privilege levels from user mode to supervisor mode is to branch or call through a special type of descriptor known as a *call gate* that points to another descriptor whose privilege level is at the supervisor level of 0. We have avoided opening a system protection hole by

- preventing direct access to the LDT by placing it in the microkernel's memory.
- restricting the `i386_set_ldt` interface only to set up descriptors at the user privilege level of 3. Any attempts to create descriptors at a different privilege cause a `KERN_INVALID_ARGUMENT` to be returned.
- omitting the call gate that implements microkernel trap access from the LDTs constructed this way. This prevents a virtual protected mode thread from altering the call gate into the microkernel to point to some random address in kernel space.

### 7.6. Memory Reservation

Both the DPMI 1.0 specification and the semantics of OS/2 shared memory[6] require shared memory regions to begin at the same address in every MVM virtual machine or OS/2 process. However, the microkernel may allocate memory for out-of-line IPC messages anywhere in the receiving task's address space, conflicting with this requirement. Hence, in order to implement the correct shared memory semantics for both DPMI and OS/2, our version of the microkernel provides a new virtual memory call and new a virtual memory protection type that allow programs to reserve address ranges in a task's virtual address space that can only be allocated by specifying a starting address within the range and which are not used in anywhere allocations. The new microkernel primitive is `vm_reserve` which provides the address range reservation mechanism. The new protection type is `VM_PROT_RESERVED`. Virtual memory calls such as `vm_map` and `vm_allocate` use `VM_PROT_RESERVED` memory only when passed a specific address at which to allocate or map.

## 8. Implementation Status

Due to the nature of the Intel legacy software, one of the difficulties with a project such as ours is determining when one is done: there is no systematic way to test conformance, and the only way to proceed is to run large numbers of applications and fix the problems

encountered. As of late February, 1993, we have a reasonably complete and well-tested implementation that

- is able to boot and run all versions of IBM PC Dos from 3.3 to 5.0
- can run a large number of DOS applications and benchmarks ranging from entertainment software like Microsoft Flight Simulator to Dos compatibility tests and benchmarks like the *PC Week* and *Byte* benchmarks.
- executes a wide range of standard mode Windows 3.0® and Windows 3.1® applications like Lotus® Amipro™, PFS Winworks™, Excel V4.0™ and Picture Publisher™.
- supports a large number of protected mode extended Dos applications that conform to the DPMI like Lotus 123 R3.1.
- runs multiple full screen Dos and Windows sessions along with a full screen OS/2 personality session running the OS/2 Presentation Manager with its Workplace Shell and a Unix personality running X11R5 and Motif®. We are able to toggle between these sessions.
- runs multiple text mode Dos applications running windowed under X11R5.
- is able to invoke Dos sessions from the OS/2 and Unix personalities.

## 9. Performance

We would like to stress that up to this point our primary goal has been to implement enough of our design to be able to do some performance measurements and analysis, and we have only recently reached the point where serious performance work can begin. The following, therefore, represents some high-level observations and simple measurements rather than detailed performance work.

We are about 20 to 30 times slower in handling Dos or BIOS services that need not be emulated than native Dos. This is indicated in item 1 in **fig 7**. The primary reason for this phenomenon is that when these services are invoked the microkernel is still generating an

Q	Benchmark/Test	DOS 5.0 Windows 3.1	MVM
1	Video.Exe, iterations=60000 This uses the Int 10H interface to put characters on the screen.	4.833 sec	60.888 sec
	Intevect.Exe iterations=60000 This uses the Int 21H interface to get an interrupt vector from the IVT.	1.444 sec	55.111 sec
	memory.exe iterations=60000 Uses Int 21H to allocate and free paragraphs of memory.	4.55 sec	117.611 sec
2	PC Tech Benchmark Wordperfect 5.1 BAPCO Byte Benchmark	34.89 110.870 secs 1.49	60.87 300.7 secs 0.72
3	Winbench ver 3.1	4,239,099	3,368,77

fig 7



exception message to the VMM rather than vectoring directly to the interrupt handler on the target thread in the virtual machine. This is the primary motivation for the addition of `i386_INTERRUPT_BITMAP` support in the kernel. We believe that we will be able to eliminate most of the overhead that we observe now by using this new microkernel feature.

We are about twice as slow as native Dos on raw disk access performance. This is indicated in item 2 of **fig 7**. To solve this problem we plan to emulate the INT 21H file system access for both virtual 8086 and virtual protected mode rather than emulating at the INT 13H BIOS entry the way that we do currently. We also plan to do some aggressive caching in our file system emulation code and to use the memory mapped file support that has been added to the file server that MVM uses. Unlike real mode Dos programs, extended Dos programs tend to use large buffers for file reads and writes using the INT 21H interface. By doing all of these things, we think that we can reach close to native file system and disk performance for extended Dos programs and environments including Windows 3.x.

We have observed that the MIG stubs seem to add overhead by doing some unnecessary copies of data. We are currently working on a separate project to improve MIG and IPC performance and expect that this will solve this problem. We have the option of hand coding certain message interfaces but plan to avoid this if at all possible.

Having said all of this, the user-perceived response of the system is fairly satisfactory as indicated by item 3 in **fig 7**. We find this very encouraging given the amount of performance work that we have done to date.

## 10. Relationship to Previous Work

In many ways our work is a logical extension of [4], and many of our ideas for extending the microkernel are based on some of their extensions including the management of the interrupt flag and the interrupt delivery mechanism. Moreover, our basic emulation mechanism, the Mach exception facility, is the same as that used in [4]. What we have done is to take the next step to extend the environment to support the very large class of Dos extenders and extended Dos applications. We have also been more careful to support the features of the real mode environment faithfully to permit a larger class of programs to run.

Our VDD model is derived from the VDD model of OS/2 2.0. We have consciously tried to provide the same interfaces and kinds of VDH services as are found in OS/2 to make it easier for implementers to port OS/2 VDDs to MVM.

## 11. Conclusions

Although we have some performance problems to fix and must provide a seamless environment for graphical applications, we are very pleased with our results. With relatively minimal changes to Mach, we have been able to implement a reasonably complete Dos and extended Dos programming environment. We found that

- small Mach changes help performance
- small Mach changes were required to get all the function needed, especially in support of DPMI
- there are many details if one is to get a complete Dos, Windows and DPMI environment
- there are some inefficiencies in the MIG stubs that need attention.

Our work was influenced, supported and assisted by many people both inside and outside IBM. We thank them for their help.

## References

- [1] David Golub, Randall Dean, Alessandro Forin, and Richard Rashid, "Unix as an application program", *Proceedings of the 1990 Summer Usenix*, Usenix Association, June 1990.
- [2] Richard Rashid, Robert Baron, Alessandro Forin, David Golub, Michael Jones, Daniel Julin, Douglas Orr, and Richard Sanzi, "Mach: A Foundation for Open Systems", *Proceedings of the Second Workshop on Workstation Operating Systems*, pages 109–113. IEEE Computer Society, September 1989.
- [3] Trusted Information Systems, Inc., *Trusted Mach System Architecture*, Internal Report, April 1990.
- [4] Richard Rashid, Gerald Malan, David Golub, and Robert Baron, "DOS as a Mach 3.0 Application", *Proceedings of the Second Usenix Mach Symposium*, November 1991.
- [5] Daniel P. Julin, Jonathan Chew, Mark Stevenson, Paulo Guedes, Paul Neves and Paul Roy, "Generalized Emulation Services for Mach 3.0 Overview, Experiences and Current Status", *Proceedings of the Second Usenix Mach Symposium*, Usenix Association, November 1991.
- [6] James M. Phelan and James W. Arendt, "An OS/2 Personality on Mach", to appear in *Proceedings of the Third Usenix Mach Symposium*, Santa Fe, April 1993.
- [7] Pylee Lennil, "DOS – A Look under the Hood to See How It Spins", *Personal Systems*, Issue 3, 1990.
- [8] Wiecek C.A., Kaler C.G., Fiorelli S., Davenport W.C., Chen R.C. "A Model and Prototype of VMS Using the Mach 3.0 Kernel", *Usenix Workshop Proceedings on Microkernels and Other Kernel Architectures*, Seattle Washington, pp 187 – 211, April 27–28, 1992.
- [9] *IBM Personal System/2 Hardware Interface Technical Reference*.
- [10] *IBM Personal System/2 and Personal Computer BIOS Interface Technical Reference*.
- [11] David B. Golub, Guy G. Sotomayor, Jr. and Freeman L. Rawson III, "An Architecture for Device Drivers Executing as User–level Tasks", to appear in *Proceedings of the Third Usenix Mach Symposium*, Santa Fe, April 1993.
- [12] Expanded Memory Specification (EMM), ver 1.x.
- [13] *eXtended Memory Specification (XMS)*, Ver 2.0. Lotus, Intel, Microsoft and AST, 1988.
- [14] *Dos Protect Mode Interface (DPMI) Specification*, Ver 1.0. DPMI Committee, 1991.

IBM, Microsoft Windows, Intel and Lotus are registered trademarks of IBM, Microsoft, Intel and Lotus, respectively. OS/2 is a trademark of IBM. Unix is a registered trademark of Unix System Laboratories, Inc. Amipro is a trademark of Lotus. Excel V4.0 is a trademark of Microsoft. Picture Publisher is a trademark of Micrografx. PFS Winworks is a trademark of PFS. Motif is a registered trademark of the Open Software Foundation.

# AN OS/2 PERSONALITY ON MACH

*James M. Phelan  
James W. Arendt  
Gary R. Ormsby  
International Business Machines  
Austin, Texas 78758*

*Internet: arendt@gunfight.austin.ibm.com*

## Abstract

Implementing a microkernel based operating system has been a hot topic in operating system research. This paper discusses the results of our work in creating an OS/2<sup>®</sup><sup>1</sup> personality server on Mach 3.0, a message based microkernel. The Mach microkernel was initially implemented with a BSD Unix<sup>®</sup> personality server.

Creating the OS/2 personality presented several new design and performance issues because the OS/2 personality and object format are drastically different from Unix. The OS/2 personality was designed to work as part of a multi-server system, containing other personalities (DOS and Unix) as well as personality neutral servers. In particular, the file system was separated into a personality neutral server.

This paper presents a brief description of OS/2, followed by a description of our server's design, a discussion of several distinctive features in our implementation, and concluding with a discussion of the current status and performance of the OS/2 personality.

## 1. Introduction

IBM<sup>®</sup> is developing a multi-server architecture for emulating multiple operating system environments concurrently using Mach 3.0, the microkernel created by Carnegie Mellon University [7,8]. The server on a microkernel concept consists of splitting a traditional monolithic system such that critical shared system and hardware dependent software portions reside in the microkernel and the remaining software moves into the user space as a single server [2, 7]. The multi server concept further divides a single server into multiple servers [4]. Some of these servers are operating system independent and some contain operating system semantics. The prior will be referred to as utility servers and the latter will be referred to as personality servers. The prototype architecture is intended to have an OS/2 personality, a Multiple Virtual Machine (MVM) personality [5] and an AIX<sup>®</sup> personality with several utility servers such as file management, communication and validation servers. The OS/2 personality presents the same Application Programming Interface (API) to an application as the OS/2 2.x product to support existing and newly developed OS/2 2.x applications.

The approach is intended to allow multiple installable personalities, provide portability on hardware platforms including multiprocessors and offer a consistent environment for developing and debugging application programs and new servers.

1. AIX, IBM, OS/2, Presentation Manager, PS/2, RS/6000 and Workplace Shell are registered trademarks of the IBM Corporation.

Unix is a registered trademark of Unix Systems Laboratories, Inc..

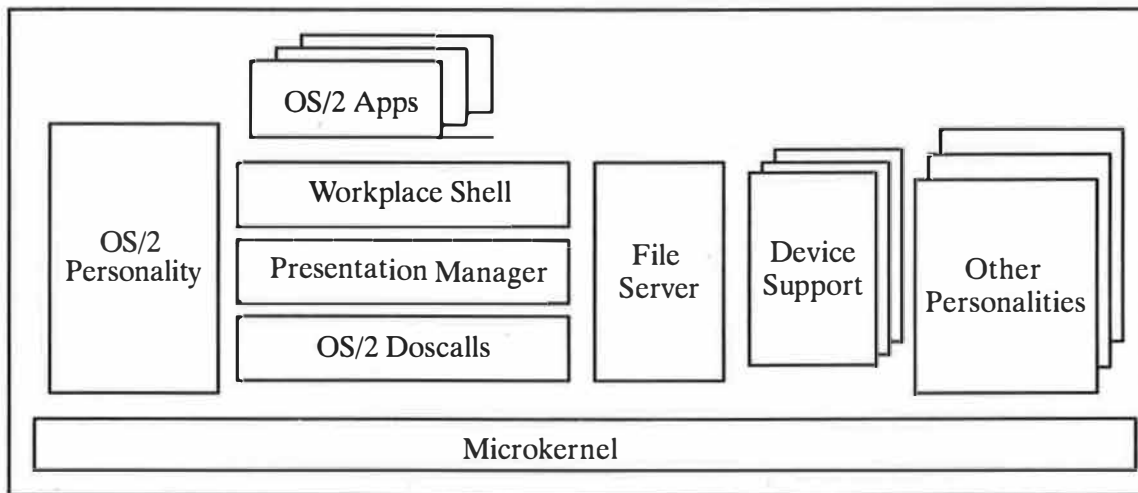


Figure 1: OS/2 Personality on Mach

The multi-server prototype was divided into three cooperating personalities projects: an OS/2 personality prototype, an MVM prototype and an AIX personality prototype. Joint efforts focused on prototyping a personality neutral file server and on developing user-level device drivers. The goals of the OS/2 prototyping project were as follows:

- Evaluate writing a non Unix based personality server using Mach 3.0.
- Support the OS/2 2.0 graphical interface, called Presentation Manager® (PM), and the Workplace Shell® virtual desktop.
- Provide binary compatibility for flat 32-bit OS/2 2.x applications on Intel platforms.
- Provide support for portability and multiprocessing.
- Co-exist in a multi-server environment with other personalities.
- Determine the characteristics and performance of the OS/2 personality server.

Figure 1 shows the OS/2 personality server in a multi-server Mach environment. This paper focuses on the Mach related issues encountered while creating the OS/2 personality prototype.

## 2. Description of OS/2

The OS/2 2.0 product is readily available and described in detail in various literature.[1,6] OS/2 has evolved from a 16 bit system written to overcome some of the perceived shortcomings of DOS into an advanced 32 bit operating system called OS/2 2.0, selling several million copies within the first year of release. The control program of the OS/2 2.x system provides preemptive multitasking, multithreading, protected virtual memory, run-time loaded shared libraries, interprocess communication in the form of shared memory, semaphores, pipes and queues, installable file systems, error and exception handling, timing functions, device support, message and national language support, and debugging support.

On top of the base services is a graphical interface called the Presentation Manager that gives a windowed, device-independent presentation space. A virtual desktop called the Workplace Shell provides an object-oriented, drag-and-drop interface to the user. Examples of available OS/2 2.0 extensions include networking and multimedia support.

To narrow the scope of this paper, we will focus only on the implementation of the server that supports the base OS/2 2.0 control program 32 bit application interface. The 32 bit interfaces were designed to be portable[1]. File systems are managed by a separate universal file server. Device support is controlled by utility servers for user level device drivers[3]. Support for DOS and extended DOS applications such as Windows 3.1 is handled by a MVM server. Of the remaining areas, certain components are of special interest as they bring to light issues relating to Mach. These include memory, loader, tasking, exceptions, semaphores, and pipes.

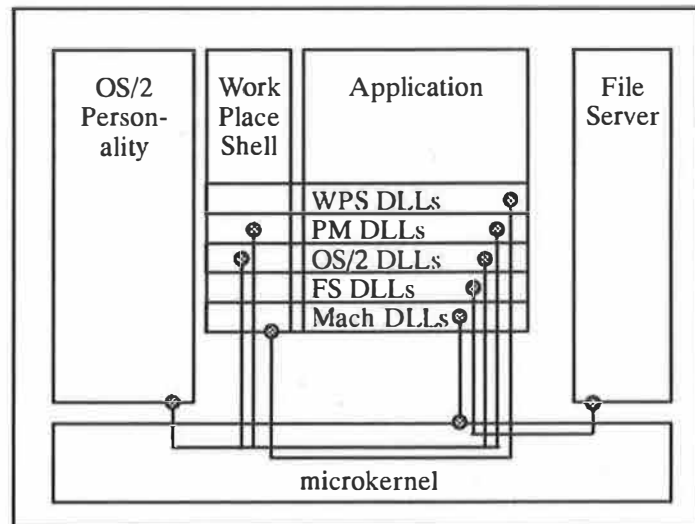


Figure 2  
Application Interfaces

## 2.1. Memory

There are three aspects of the OS/2 memory semantics that are of particular interest with regards to Mach. They are the shared memory model, the guard page model, and the committed memory concept. The OS/2 memory model includes private and shared memory objects, four classes of protection, and sparse (uncommitted) memory. The model for sharing memory includes named and unnamed (anonymous) objects and requires that all instances of shared memory objects have the same virtual address.

The OS/2 memory model has two uses of guard pages. The first and most common usage is for the dynamic growth of stacks. The second usage allows an application to detect references to or execution of a page of data or text. In either case any reference to a guard page causes an exception to the system. If the application does not have an exception handler, the default action is to mark the page as a normal page and allocate the next stack page and mark it as a guard page. If the latter fails, the application is issued an *Unable to grow stack* exception.

The concept of committed memory allows a user to allocate sparse arrays without over committing swap space and to avoid having a user of an application lose his work due to her system crashing because the swap space was over committed. Swap space is committed for an application to avoid unrecoverable system errors prior to saving end user's updates. The basic idea is that the system keeps count of how much memory has been allocated and any new allocation that would over commit the swap space is rejected and the application program can do whatever is appropriate.

## 2.2. Loader

Program loading in OS/2 follows the memory model discussed above and depends on the generic .EXE format of executable files. The OS/2 2.x loader can load several 16 and 32 bit variations of the .EXE format. The most current is called the Linear eXecutable format or LX. Each of these variations has a dynamic loading capability called Dynamic Link Loading (DLL) which defers the actual linking to a library until execution time. By implementing the applications interface to the system in DLLs, the implementation of these API can be changed

without altering existing executable binaries. OS/2 takes advantage of this aspect to allow parts of a system call to execute in the user's address space. When an end user upgrades to a new system, the user replaces the kernel and the system DLLs.

### **2.3. Tasking**

The model for spawning a process in OS/2 creates clean copies of the address space and inherently provides a multi-threaded execution environment. The environment, argument list and certain handles are inheritable. The loader and DLLs discussed above are used to bring the program into memory and prepare it for execution. Programs can be loaded, started synchronously or asynchronously, or placed under debugging control.

A running program can create new threads to share process resources. Each thread has a unique register context and stack. Threads can be terminated, suspended or resumed. A thread can also wait for the termination of another thread or process. Process exit lists contain routines to be run when a process is terminated or ends its execution.

The OS/2 2.x product contains an elaborate scheduler. To the user, there are four priority classes (time-critical, server, regular and idle), each with 32 priority levels. The time slice and starvation interval are configurable by the user at boot time. The priority boosts vary by class and include boosts for input focus, screen foreground, I/O completion, and starvation.

### **2.4. Exceptions**

Exceptions occur on a per thread basis, and the user can register exception handlers for a thread. If no exception handler is registered, a system default action is done. Nested exceptions are supported. Certain events external to a thread can cause special exceptions called signal exceptions, including break, interrupt or kill process exceptions. The signal exceptions were managed by a separate signal interface in earlier versions of OS/2. Asynchronous exceptions, those caused by external events including signal exceptions, can be deferred by specifying a must complete section.

### **2.5. Semaphores**

OS/2 has three types of semaphores. Event semaphores notify waiting threads when a specific event has occurred. Mutex semaphores control access to shared resources and have the concept of exclusive ownership. Finally, muxwait semaphores wait for multiple events to occur or for multiple resources to become available. Alternatively, a flag can be set so that the semaphore will indicate when one of multiple events has occurred or when one of several resources has become available. All of the semaphores utilize timeouts which allow a thread to resume execution if it is blocked too long while waiting for a semaphore. Infinite timeouts are possible.

Event semaphores are used to ensure that events happen in a desired sequence. They have two states, reset and posted. When in the reset state, an event semaphore will cause any thread waiting on it to be blocked. When in the posted state, all threads blocked on the semaphore will resume execution.

When a thread wishes to use a shared resource protected by a mutex, it requests ownership of the semaphore. If the semaphore is currently owned, the thread will block until the semaphore is released. Otherwise, the requesting thread gains ownership of the semaphore. Ownership is maintained until the owning thread releases the semaphore. At that point, the thread with the highest priority obtains ownership of the semaphore. Mutex semaphores are typically used to regulate access to critical sections of code.

Muxwait semaphores consist of up to 64 event or mutex semaphores (the two types cannot be mixed on the same muxwait). When the muxwait is created a flag is set to specify that the semaphore is used in one of two ways.

1. Waiting threads are released when all of the mutex semaphores on the muxwait are released or when all of the event semaphores are posted.
2. Waiting threads are released when any of the mutex semaphores on the muxwait are released, or when any of the event semaphores are posted.

Semaphores can either be named or anonymous. Named semaphores are always shared, which means they can be used by any process that knows the semaphore's name. Anonymous semaphores may or may not be shared, depending upon the value of a flag when the semaphore was created. Shared semaphores are opened by processes before they are used. In the case of named shared semaphores, the name is passed to the open routine. Anonymous shared semaphores are opened by passing the semaphore's handle. Anonymous private semaphores only can be used by the process that created them.

## **2.1. Pipes**

OS/2 pipes are named or unnamed circular buffers. Unnamed pipes are always duplex and allow related processes to share data. Named pipes are more complex. They can be input-only, output-only, or duplex. They can transfer data as bytes or as messages. Additionally, named pipes can have several instances which allow a single server process to communicate with several client processes.

Unnamed pipes have a write handle and a read handle. Typically a parent will create an unnamed pipe, duplicate its handles as standard in and standard out, and spawn a child. The child will inherit the parent's handles and communicate with the parent by writing to standard in and standard out.

Named pipes allow related or unrelated processes on either the same computer or different computers to communicate with each other. A process that knows the name of the pipe can open it and use it. The process that opens a named pipe is known as the server process and it controls access to the pipe. Client processes open the named pipe and are connected to the other end from the server. The server receives a handle when it creates a named pipe and clients receive a handle when they open the pipe.

When a named pipe is created by the server process the attributes of the pipe are specified. The server specifies if the pipe will be one-way or duplex, whether it will be a byte pipe or a message pipe, and the maximum number of clients that can open the pipe. Named pipes can either be byte pipes or message pipes. Byte pipes have individual bytes read and written to them. Message pipes operate on whole messages. The size of the messages are specified at pipe creation time.

## **3. OS/2 Personality Server Design**

The OS/2 personality server exists as a server and a set of dynamic linked libraries that are mapped into the application's address space. The dynamic link method routes application system calls into the dynamic linked libraries using loader fixups at execution time. The libraries either handle the call themselves or forward the request using a Mach message to the OS/2 personality server or another server.

The server contains a message loop that receives these messages and calls the appropriate sub-routines to satisfy the user's request. The port on which the message is received is used as a handle for the object to operate on, using a hash table to map the port to a typed object address. The OS/2 personality server also responds to external paging requests, application exceptions, and control functions.

The OS/2 personality server was developed using the libmach and cthreads libraries. The server is written in C. The Mach interface generator (MIG) was used to create remote procedure call interfaces. The BSD Unix server was used to bootstrap the OS/2 personality server, illustrating the advantage of a user space server that can be debugged like other applications. However, the BSD Unix server need not be present.

## 4. Distinctive Features

This section discusses Mach issues of general interest encountered while developing the OS/2 personality server. The issues are broken down into the specific components including memory, loader, tasking, exceptions, semaphores, pipes, timers and files.

### 4.1. Memory

The private and shared memory model required specific address ranges be reserved for each OS/2 application. Some of the shared memory is shared between all tasks, some is shared only between selected tasks, and some requires different copies but at the same virtual addresses. For example in Figure 2, **Task 3** does not have named **shared area b**, but the region of virtual memory must remain un—allocated in case **Task 3** maps **shared area b** in at a latter time. Implementation of this requirement using regular Mach was not possible because Mach assumes out of line memory can appear anywhere in the address space. The solution to this problem was an enhancement to Mach called address space reservation. As a user task is created and before the task is started, the server requests the region of the user's address space that will be used as shared memory be reserved. The kernel then avoids allocating memory in that region except for a specific request. This was a fairly straight forward alteration that required a new Mach call *vm\_reserve* and subtle changes to other memory calls (e.g. *vm\_region*) to account for the new class of memory.

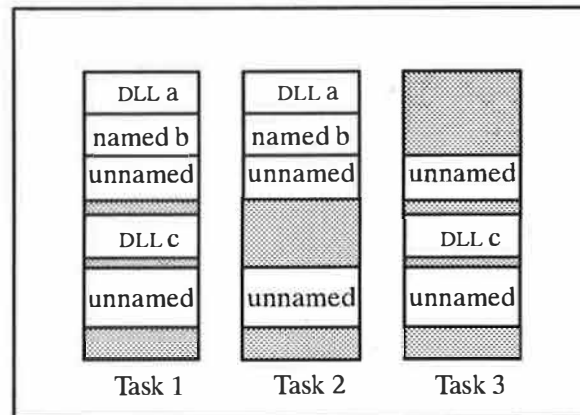


Figure 3  
Shared Memory Classes

The implementation of guard pages presented another interesting problem. An inherent design assumption of the OS/2 server was to avoid duplicating information that was already maintained by the microkernel. Instead effort was expended to determine how to complement the microkernel's functionality. For memory services this meant keeping track of the special object attributes and not keeping track of memory page states in server data structures. As a result the two uses of guard pages have been implemented differently. Guard pages that are used for stacks have no useable data (zero filled) and are implemented with a special memory object called the guard pager that always emits an error response with a magic cookie as the error code. The error results in a memory exception exception into the server. The server detects the guard page status and expands the user stack. In earlier versions of the microkernel, the magic cookie was passed to the server in the generated exception, which made detection of guard pages easier. But this has changed in more recent versions and the server now makes a *vm\_region* request to detect the guard pager. For the second usage of a guard page mentioned above, the guard page contains valid data and must be cached. This



is done as a duplicate copy on write page but still requires the server to keep track of where the page is cached by the server.

The initial design of the prototype and the current implementation keep track of the amount of committed memory required by all the OS/2 server's applications through calls to a single routine. This routine maintains the amount of request swap space relative to a dynamic water mark, but does not actually pre-allocate swap space. The pre-allocation of swap space within a multi-server environment was viewed as a larger problem than could be managed by a single server such as the OS/2 server. As such, the problem was 'kicked upstairs' and the current prototype can be readily enhanced to use the new *memory\_object\_data\_allocate* system call.

## **4.2. Loader**

The prototype's loader is designed on a lazy evaluation principle. System DLLs tend to reference other DLLs creating trees of DLLs. The implicit assumption is that the trees of included DLLs can be quite large and that a large number of these DLL objects will not be used in a standard execution. The projected result is that allocating a multitude of pagers for objects would cause the allocation of a multitude of Mach ports along with a multitude of messages to initialize the pagers, and the mapping of a multitude of memory objects would cause a multitude of memory ranges in the kernel's data base. Together, these will tend to clog the kernel's tables and degrade performance. Thus the loader does not assign addresses to any DLL objects until they are referenced, much like faulting objects into memory. The loader will not map an object into a user's space until the user references the object, also like faulting in objects into a user's address space. Note that all memory pages are faulted into memory and that no memory objects are pre-loaded.

In the prototype, the general flow for loading an application is to map in all the objects from the application's .EXE file, the DLL objects the application directly references and the DLI's initialization objects. All the user's object must be mapped into memory because the user can specify the execution starting address for his program at any location and not just the starting address found in the binary. The remaining DLI's are not allocated memory as described above. The user is then started and demand loads pages as they are required. As the page fault occur, the memory section (a pager) receives the faults from the kernel and determines the faulting page's location in terms of which page of which object belonging to which MTE. The memory manager then requests the required page from the loader. As the loader reads in a page and fixes it up, it detects referenced objects that have not been assigned an address in the shared arena. The loader then requests a memory slot of the proper size in the shared arena and maps the object into the faulting user's address space. The user is then restarted and continues. When another task references the page, the kernel will give him a copy of the the page that the kernel has cached and avoids requiring the loader to fix up the page. Because the loader dose not fix the page on subsequent references, the user does not have the objects that are referenced on that page mapped into his address space. When the user hits one of these references, a memory protection fault is returned to the exception portion of the OS/2 server. In processing this exception, the server detects the absence of the mapping for the task and corrects the situation by mapping the object into the task and restarts the user task -- faulting the object into the user's address space. The user is then restarted as above.

## **4.3. Tasking**

In the current OS/2 personality server, OS/2 tasks and threads are mapped one-to-one with Mach tasks and threads. The tasks and threads can be suspended, resumed or terminated

using the Mach kernel interfaces. Concerns that arose involved fast thread identification, handle management, scheduling and critical sections.

The OS/2 stacks can be allocated and controlled by users, requiring a novel way of quickly identifying threads. The `cthrads` package uses the stack pointer and controlled placement of the user stacks for fast thread identification. The kernel call `mach_thread_self()` gives a send right to the thread port that can be used to uniquely identify the thread but this requires an additional kernel call. Platforms such as the IBM RS/6000® contain a regular register that can be used to store the thread identifier. On the Intel platform, since the memory model is flat, an unused segment register exists. This fs segment register is used to identify the thread. The nature of segment registers on the Intel 386 platform requires that the segment register points to a valid descriptor table entry. The Mach kernel was extended to provide valid thread identification descriptors on the Intel 386.

Handles for files, pipes and devices are represented by Mach ports with a table in the application space mapping the conventional values to the corresponding ports. Since handles in the handle table can be inherited, the tasking component needs to transfer additional copies of these send rights to the newly created task. Because Mach does not support inheritance of port rights, each right needs to be placed in the new address space by an additional kernel call or sent in a message and this slows down task creation.

The current Mach kernel scheduler provides only 32 priority levels with either fixed priority or variable priority with starvation boosts. The OS/2 personality prototype maps the OS/2 priority levels to Mach priorities in a simplistic way, losing some of the abilities of the original product. In addition, the issue of the preferred method of scheduling threads between multiple servers and multiple personalities has not been resolved.

OS/2 has the notion of critical sections, which suspend execution of all other threads in a process until the critical section is complete. To implement this call on Mach, each other thread in a task is suspended individually using a kernel call when the critical section is entered and resumed when the critical section is exited. Suspending the entire task prevents the thread in the critical section from running. The individual calls make a critical section very expensive compared to a system such as the OS/2 2.x product that contains scheduler support.

#### **4.4. Exceptions**

The exception component handles Mach exceptions by setting the task exception port for an OS/2 task to be send rights to a port in the OS/2 personality server. When an exception occurs, a message is sent by the Mach kernel to this port. The exception is mapped to the corresponding OS/2 exception, the context is recorded, and existing exception handlers are run or the default action taken. The task exception port is used to allow knowledgeable users the ability to handle Mach exceptions directly using the thread exception port.

#### **4.5. Semaphores**

There are two points of interest in our implementation, blocking a thread's execution and timeouts. OS/2 API calls are translated to mig calls. When a thread is made to wait on a semaphore, either by the semaphore being owned or by the thread calling `DosWaitEventSem` or `DosWaitMuxWaitSem`, the thread is enqueued on the semaphore's list of waiting threads and the reply message is withheld. The invoking thread's reply port is saved along with a pointer to the thread's internal data structure in the semaphore's list of waiting threads. When a semaphore becomes available, one or more threads resume execution by having their records removed from the semaphore's list of waiting threads and by sending the appropriate mig reply to each of threads' reply ports.

Timeouts also work on the principle of withholding a Mach message's reply until the invoking thread can resume execution. When our OS/2 server is started, a thread is spawned that monitors the timeout queue. This is a queue of threads that are waiting on semaphores with finite timeout values. The queue is organized in ascending order. A dedicated port is allocated for use by the timeout mechanism.

When a thread requests a semaphore with a finite timeout value, a pointer to the thread's structure, the timeout value, the semaphore's type, and the thread's reply port are added to the timeout queue. When a request is placed in the front of the queue, a *mach\_msg* is issued to receive a message from the timeout port with a timeout value equal to the request's. When the *mach\_msg* times out, the first element is removed from the timeout queue and a mig reply is sent to the newly dequeued thread's port. A different reply is sent for every type of semaphore since each has its own wait semantics.

#### 4.6. Pipes

Our server's pipe component implements both unnamed and named pipes. It uses Mach specific technology for several things. Each pipe handle has a port associated with it. By giving out port rights only to tasks that create or open pipes we insure that only those tasks with send—rights can access the pipes they rightfully are entitled to. Named pipes also have a timeout mechanism that is very similar to the semaphore mechanism.

When a pipe or named pipe structure is allocated, either by creating or opening a pipe, a pipe structure and a port are allocated. The port is associated with the structure and is stored in a global handle table along with the address of the pipe's structure. The port is also placed in the task's handle table and a handle is returned to the user. When a pipe API call is made, the user's handle is translated to the pipe's port stored in the task's handle table. The port is passed in a mig call to the OS/2 server and is translated to the address of the pipe's structure by finding the port in the global handle table.

Since all APIs are translated to mig calls, a task's authority to access a given pipe is verified by the Mach kernel when the mig call is sent to the OS/2 server. Any process making a pipe API call must have a send—right for the pipe's port. Similarly, when a handle is duplicated, another entry is made in the user's handle table and the send—right count for the pipe's port is incremented by 1.

Data that is read from or is written to pipes is passed between the user—side and the server—side out—of—line. This favors the transfer of larger data blocks since only a page table manipulation is required. Optimally the choice of passing data through mig should be made at run—time, based upon the size block being transferred.

No—senders notification is set when a port is associated with a newly created pipe structure. Closing a pipe deallocates a send—right for the pipe's port. When the send—right count reaches 0, the no—senders notification calls a routine which deallocates the pipe structure associated with the pipe. Note that if a pipe handle is duplicated, all handles must be closed before the no—senders notification triggers the pipe's deallocation.

#### 4.7. Timers

The OS/2 control program provides timer functions that allow the application thread to sleep for a period of time or for an event semaphore to be posted after a time interval. The time functions were implemented using the Mach message timeout abilities. Sleeping becomes a message receive on a port with a timeout value. The sleeping thread can be wakened early by sending a message to the port. For the timer functions that post semaphores, a thread in the

server keeps a time-ordered queue and sleeps using the message timeout until the next timer interval expires.

#### **4.8. Files**

Files in the current scheme are no longer owned by the OS/2 personality. Instead, we have provided a personality neutral file server. The existing OS/2 file interface is provided to applications by a thin layer in a DLL in the user space. Actual file requests are mapped to the personality neutral file server interface. Mach ports are used as mount points and to represent files, and are managed by the DLL in coordination with the OS/2 personality server.

Since the file server is shared between personality servers and is designed to be extensible, this scheme quickly allows for sharing of FAT, HPFS, and Unix style journaled systems. Capabilities such as memory mapped files can also be developed and shared across personalities.

### **5. Status and Performance**

The OS/2 personality has been demonstrated running the Workplace Shell and PM applications on the floor of the Fall 1992 COMDEX show in Las Vegas. The current implementation co-exists with MVM and Unix personalities, using a personality neutral file server and user-level device drivers.

Very little performance tuning has been done at this stage. For example, the code is not yet compiled with optimization on to simplify debugging. An unoptimized version of the file server that does not support memory-mapped files is used. The tuning process is causing frequent changes in the performance characteristics of the OS/2 personality. Current performance numbers will be presented at the Third Usenix Mach Symposium.

### **6. Conclusions**

We discussed our implementation of a prototype of an OS/2 personality using the Mach 3.0 microkernel. The OS/2 personality has been demonstrated running such complicated 32 bit OS/2 applications as the Workplace Shell and supports the PM graphical interface.

Mach provides an adequate framework for an OS/2 personality server, again demonstrating its usability in a non-Unix environment. The kernel needed to be extended to support the reservation of virtual memory addresses. The current microkernel scheduler lacks the capabilities of the OS/2 2.x product. And an optimized method of thread identification on an Intel 386 or 486 processor made necessary changes for setting up valid segment descriptors. In general, however, the current Mach kernel interface is sufficient for our OS/2 personality prototype. Other personalities and utilities servers, including those used by the OS/2 personality, have their own microkernel requirements.

Developing the OS/2 personality server on Mach provides some obvious benefits. The new server was debugged like other Mach applications, easing development. The microkernel has demonstrated portability and contains support for multiprocessor environments. This support is being used as a basis for providing multiprocessor support and source-level OS/2 compatibility on other hardware architectures using the OS/2 personality server. The OS/2 personality can co-exist with other Mach tasks such as the MVM and AIX personality servers and common utility servers, with these other servers run in their own protected address spaces.

We appreciate all the assistance received from people in support of this project and extend our gratitude to them.

## References

- [1] Deitel H. M., Kogan M. S. *The Design of OS/2*, Addison Wesley, 1992.
- [2] Golub D., Dean R., Forin A. *Unix as an Application Program*, Usenix Summer Conference, June 11–15, Anaheim California, summer 1992.
- [3] Golub D., Sotomayor G. G. Jr., and Rawson R. L. III *An Architecture for Device Drivers Executing as User-level Tasks*, to appear in the Proceedings of the Third Usenix Mach Symposium, Santa Fe New Mexico, April 1993.
- [4] Julin P. J., Chew J. J., Stevenson J. M., Guedes P., Neves P., Roy P. *Generalized Emulation Services for Mach 3.0, Overview, Experiences and Current Status* Usenix Symposium Proceedings on Mach, Monterey California, November 20–22, 1991.
- [5] Manikundalam R., Golub D., Rawson R. L. III *MVM – An Environment for Running Multiple DOS, Windows and DPMI Programs on the Microkernel*, to appear in the Proceedings of the Third Usenix Mach Symposium, Santa Fe New Mexico, April 1993.
- [6] *OS/2 2.0 Technical Library*, IBM Corporation, 1992.
- [7] Rashid R., Baron R., Forin A., Golub D., Jones M., Julin D., Orr D., and Sanzi R. *Mach: A foundation for open systems*, Proceedings of the second workshop on Workstation Operating Systems, IEEE Computer Society, pp 109–113, September 1989.
- [8] Silberschatz A., Peterson J., Galvin C. *Operating System Concepts*, third edition, Addison Wesley, 1990.
- [9] Young M. W. *Exporting a User Interface to Memory Management from a Communications-Oriented Operating System* PhD Dissertaion, School of Computer Science,, Carnegie Mellon University, Pitsburg Pennsylvania, 1990.
- [10] Wiecek C. A., Kaler C. G., Fiorelli S., Davenport W. C., Chen R. C. *A Model and Prototype of VMS Using The Mach 3.0 Kernel*, Usenix Workshop Proceedings on Micro-kernels and other Kernel Architectures, Seatle Washington, pp 187– 211, April 27–28, 1992.



# Page Prefetching Based on Fault History

*Inshik Song, Yookun Cho*

*Dept. of Computer Engineering, Seoul National University*

## ABSTRACT

Many programs exhibit repeated memory access patterns. Assuming this memory access behavior, we propose a page prefetching method to reduce page-in delay time. We gather page fault information into a history buffer at run time. If a page fault occurs more than once for a given page, the pager prefetches the next page faulted following that page in the history. Using only 1% of the total free page frames for page fault history and page prefetch buffers, we have observed a high fault-again ratio of about 70% and a prefetch hit ratio of about 50% across all page faults. This historical page prefetching reduces page-in delay and increases paging performance by 20%. This method may be useful in all demand-paging systems, especially for large vector applications such as matrix multiplication.

## 1. Introduction

Modern computer systems use demand-paging to implement virtual memory. When a program refers to data that does not exist in memory, the operating system receives control via a page fault exception. Then the operating system reads the page containing requested data from the backing store into a free page frame and resumes the faulted process. Demand-paging postpones page-in operations as long as possible to keep more page frames free, but demand-paging incurs some page-in delay. To reduce this page-in delay, some systems use sequential prefetching or clustered paging. We made the assumption that memory access patterns are repeated and verified this through the analysis of page fault traces. On the basis of this result, we can predict and prefetch a page that is very likely to be accessed in the near future. The page fault handler gathers page fault information in fixed size history buffers at run time. If a page fault for the same page occurs more than once, the page fault handler looks up the next requested page in the history buffer and prefetches it into a prefetch buffer. Later, when the prefetched pages are requested, the fault handler copies prefetched data from the prefetch buffer into the requested page frame. If the prefetched page remains unrequested until all prefetch buffer are used, then it is discarded by the pager. We have tested this prefetching method on an i486/Mach system to determine the performance characteristics. Results indicate that this prefetching method is useful, especially in large vector applications such as matrix multiplication.

The remainder of the paper is structured as follows. The next section introduces the summary of the background studies and related work. Section 3 explains the page prefetching method. Section 4 presents experiments and analyzes the experimental results. Section 5 gives future work to be studied. Finally, Section 6 summarizes our results and presents conclusions.

## 2. Background and Related Work

Most computer systems have three or more levels of hierarchical storage structure including cache, memory, and disk. Information is usually transferred to higher levels on a demand basis. Cache-to-memory mapping is provided by hardware, and memory-to-disk mapping is provided by the operating system's virtual memory management subsystem with basic hardware support. Demand-paging is the most popular virtual memory management technique. In a demand-paging system, when programs access data that does not exist in memory, the operating system takes control via the page-fault exception mechanism. The page fault handler reads the requested data from the backing store into a free page frame and resumes execution of the faulted process. The faulted process must wait for the faulted page to be read in. This delay is very long compared to the CPU time. Average disk access time is at least  $10^3$  -  $10^4$  times greater than average memory access time. In a multiprogramming system, the operating system switches the CPU to a process that is ready to run in order to minimize CPU idle time. However, there is not always a program ready to run, and some idle time frequently occurs. We can use prefetching to predict the next memory reference and fetch it from the disk before it is requested to eliminate some of this idle time. Prefetching has some overhead. Although prefetching in cache systems can be supported by hardware in parallel with program execution, page prefetching has to be processed by the operating system. If most of the prefetched pages are not accessed within the near future, these prefetched pages move out other pages which contain more important data resulting in inefficiency. However, if many of the prefetched pages are accessed in the near future, reduction of the page-in delay compensates for this prefetching overhead.

Previous studies used a program's locality and sequentiality in predicting of which pages to prefetch. A program's locality has two aspects, temporal locality and spatial locality. Temporal locality means that information recently referenced by a program is likely to be used again soon. This property is expected from the fact that programs have loops. Spatial locality means that portions of the address space near the current (or recent) locus of reference is likely to be referenced in the near future. This property is again expected from common knowledge of programs: related data items are usually stored together, and instructions are executed sequentially. Sequentiality is closely related to spatial locality. The locality principle says that the most useful pages are those that have been recently referenced and are therefore already in memory. Sequentiality suggests that pages following the one accessed are likely to be referenced. Therefore, sequential page prefetching is very helpful in the initial stage of program execution. Smith reported that sequential prefetching of small data blocks is very effective for hierarchical storage machines. He pointed out that sequential prefetchings on every memory access increases effective CPU speed by 10% - 25% [Smith78]. Another study mentioned that using the partial knowledge of the future reference string in prepaging reduces the paging overhead of array algorithms operating on large arrays [Trivedi76]. Other systems use page prefetching to enhance paging performance. OSF/1's clustered paging is an example [Black91]. Prefetching is also used in a UNIX buffer cache [Leffler89]. If there is a request to read a block which is sequentially next to the previous block, the system issues a read-ahead request to the next sequential block. The buffer cache stores the recently accessed blocks for the file system. It is difficult to keep the history for the file system access because the file system contains more large data than the program's address space does. The program's memory access behavior may be different than the file access pattern. Subramanian proposed an external pager that receives information regarding discardability from the client, saves and restores only non-discardable pages and pre-flushes discardable pages [Subramanian91].



Approaches such as sequential prefetching and clustered paging are based on the locality of programs. While a number of studies about the program's memory access behavior have derived some interesting models, it is very difficult to use any one of those models in page prefetching [Denning80] [Chow76]. We assume that memory access behavior is repeated, and use this property during the execution of programs to determine which pages are to be prefetched next.

### **3. Page Prefetching Method**

#### **3.1 Fault-again and Prefetch hit ratio**

When page prefetching is performed by the operating system, it is impossible to prefetch pages for every memory access, as is the case for hardware prefetching in cache systems. We can prefetch pages only when page faults occur. The major performance factor in prefetching is the prefetch hit ratio. The prefetch hit ratio is the ratio of the number of prefetched pages actually accessed within the near future to the total number of pages prefetched. To get a high prefetch hit ratio we have to make a prediction about which pages to prefetch next based on accurate information about the program's memory access behavior. One approach is to use memory reference traces at run time. Another approach is to use reference hints given by users or the compiler [Callahan91]. We discarded the second approach for two reasons. First, it is costly for users or compilers to predict a program's memory access behavior accurately. Second, and more importantly, a process's paging behavior is largely affected by the other processes in the system and the prediction in the context of a single process may be inaccurate. Page fault traces gathered at fault time are good candidates on which to base prediction of pages to be prefetched next [Min92]. The fault again ratio is the ratio of faults that occurred more than once for the same page to the total number of page faults. For example, assume that page P' has been requested following the previous page P. With historical page prefetching, P' is the next page to be prefetched if a fault for page P occurs again. In our study, we observed 60% - 70% fault-again ratio and a 50% prefetch hit ratio for a lightly loaded i486/2.6MSD system with 8MB memory.

#### **3.2 Page Prefetch Strategy**

The structure of the page prefetch system is shown in Figure 1. History buffers are used to gather fault information at fault time. Prefetch buffers are used to hold prefetched pages until they are used or discarded. History and prefetch buffers are the major components added to enhance the performance of the paging system. Space allocated for these buffers would normally be used for free page frames if prefetching was not enabled. Therefore, these buffer sizes must be kept small enough to make more memory free, but large enough to get a high prefetch hit ratio. Many demand paging systems use a modified LRU scheme for page replacement, and the selection of pages to be swapped out is done not per process but globally. Therefore, the multiprogramming level may affect the performance of page prefetching. However, it is very difficult to model or to predict memory access behavior of multiple processes because the memory access behavior of a process is largely independent of those of other processes. We have chosen the memory object of a process to be a prefetch prediction domain. Mach memory objects represent some interprocess memory access behaviour such as sharing. History buffers are managed by the memory object and the number of history buffers is determined as a function of system memory size. Prefetch buffer space is maintained globally because a small number of

prefetch buffers is enough in most cases, especially when the multiprogramming level is low. The number of prefetch buffers is also determined as a function of system memory size.

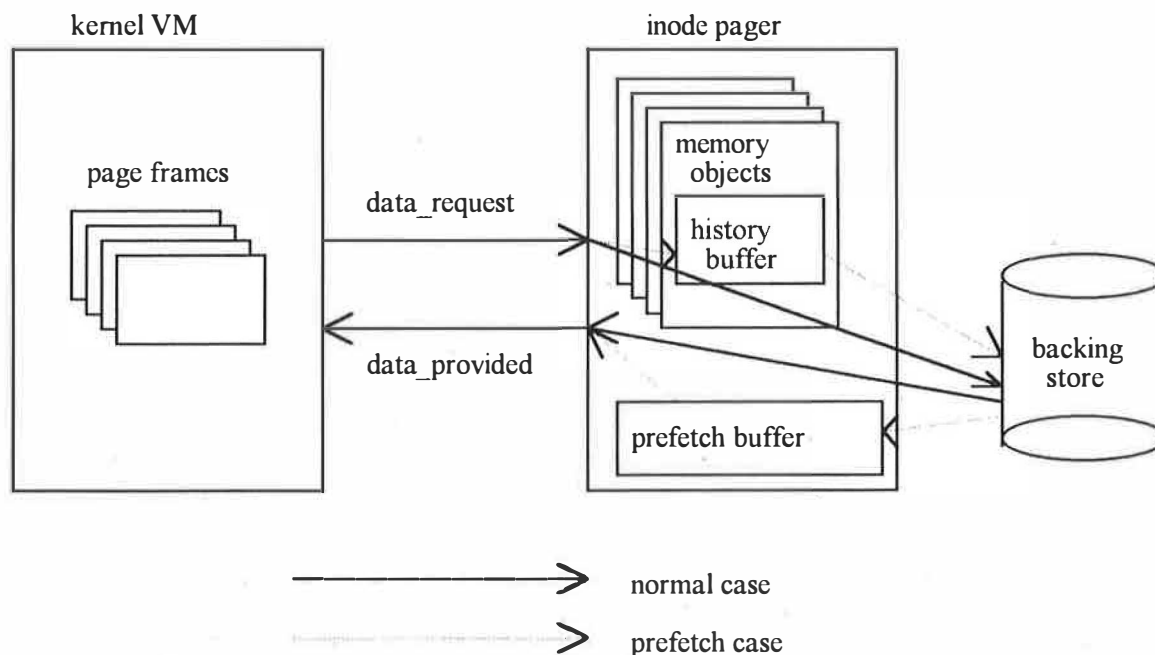


Figure 1. Structure of the page prefetching system

Figure 2 represents page prefetching algorithm. Whenever a page fault occurs, operating system's page fault handler looks to see whether the page is already in a prefetch buffer. If it is found in a prefetch buffer, the page fault handler copies the data from the prefetch buffer to the associated page frame. If it fails to find the page in the prefetch buffers, it reads the page data from the disk directly. Before resuming the faulted process the page fault handler records faulted page number in the history buffer of the memory object to which the page belongs. At this point, we check whether the page  $P$  has been faulted before. If it has, we check the page  $P'$  that had been faulted following  $P$  previously. If it is not already in memory, the page fault handler asks to prefetch  $P'$ . The page prefetching itself may be carried out during fault handling or by the independent kernel-level prefetching thread. We have tested both cases. When a page-out for page  $P$  occurs or the memory object containing  $P$  is destroyed, prefetched data is no longer useful. In these cases, such entries in prefetch buffers are discarded.

### 3.3 Performance Model of Page Prefetching

Page prefetching enhances paging performance, but it incurs some cost. It is desirable to use page prefetching in cases when the positive effect of prefetching is much larger than the negative effect. Let  $C_d$  be the cost of conventional demand-paging, and  $C_p$  be that of prefetching systems. Then  $C_d$  and  $C_p$  can be represented as following:

```

fault_handler(P)
{
    lookup prefetch buffer
    if (page P exists in prefetch buffer) {
        copy prefetch buffer into the page frame P
    } else {
        read P from disk
    }
    record fault entry into history buffer
    if (P is fault-again) {
        get P' which was requested following P in the past
        if (P' does not exist in memory)
            request to prefetch for P'
    }
}

pageout(P)
{
    invalidate prefetch buffer for the page P
}

do_prefetch(P)
{
    read P from disk into the free prefetch buffer
}

```

Figure 2. Page Prefetching Algorithm

$C_d = pf$   
 p: average page fault handling cost  
 f: page fault ratio  
  
 $C_p = pm + p'h + C'$   
 m: prefetch buffer miss ratio  
 h: prefetch buffer hit ratio  
 p': average cost for prefetch buffer hit  
 C': additional prefetch cost

Generally it is true that ( $p' \ll p$ ).  $C_p$  is determined by  $m$  and  $C'$ . Although page fault ratio,  $f(= m + h)$ , for prefetching systems, is greater than the page fault ratio for non-prefetching system, the miss ratio  $m$  decreases as the hit ratio  $h$  increases. Additional prefetch cost  $C'$  is dependent on the number of effective free page frames and the number of prefetch requests generated. As the fault-again ratio becomes higher, the value of  $C$  increases. The first term ( $pm$ ) in  $C_p$  decreases as the fault-again ratio becomes higher, because a higher fault-again ratio means a higher prefetch hit ratio. Using a low priority kernel-level prefetching thread to do prefetch

asynchronously, we get a small  $C'$ . To increase the number of free page frames, it is desirable to reduce the number of history buffers and prefetch buffers. A small number of history/prefetch buffers means a low fault-again/prefetch hit ratio. Therefore, the numbers of these buffers and buffer structures must be tuned for the target system environment through empirical tests. Another important factor in improving performance of page prefetching is the reduction of unnecessary context switching during page-in operation. Context switching is very costly because it requires a large amount of operating system intervention. A recent study pointed out that context switching in cache memory systems is much more expensive [Mogul91].

## 4. Experiments

### 4.1 Environments

Simulation and prototype evaluation are performed to examine the feasibility of a proposed page prefetching method. The target system is a i486/2.6MSD system with 8MB of memory and a 4KB page size. After successful startup of the multi-user environments, about 4MB, or 1000 pages, of memory remains as free page frames [Wang91]. If we use 4K entries of history buffers and 8 prefetch buffers, the required space to implement page prefetching is about 16 pages. It is about 1% - 2% of total available free page frames.

In the 2.6MSD system, paging is performed via a well-defined message communication protocol by the kernel VM subsystem and an inode pager that is implemented as a separate task sharing address space with the kernel [Rashid88] [Young87]. A process's address space is divided into VM objects. VM objects correspond to memory objects managed by the inode pager. The message interface between kernel VM and the inode pager is presented in Figure 3. When a page fault occurs, kernel VM sends a page-in request to the inode pager with a `memory_object_data_request()` message. The inode pager reads the requested page from the backing store and returns the contents to the kernel with a `memory_object_data_provided()` message. Pageout operation is performed by `memory_object_data_write()` message. Receiving `memory_object_data_write()` message, the inode pager writes paged-out data into the backing store.

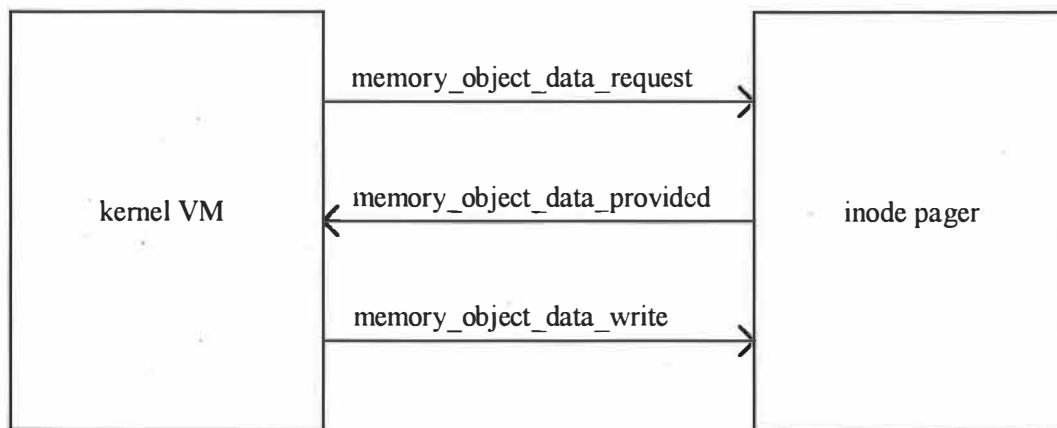


Figure 3. Message interface for paging between kernel VM and Inode Pager

## 4.2 Procedures and Results

First, we analyzed page prefetching effects under a light load condition. We gather the page fault trace during the normal operation of a i486/2.6MSD system. More than 10 users working on several projects are using this system. Major workloads are program compilation, editing, network and file access. We calculated the fault-again ratio and prefetch hit ratio with run-time page fault traces containing about 25,000 entries. We measured fault-again and prefetch hit ratio as the number of history/prefetch buffers varies. Circular queue structures are used for both types of buffers. In Figure 4-(a) we see that about 70% of faults occurred more than once when  $H \geq 4096$ . Figure 4-(b) shows that prefetch hit ratios of about 50% (70 % of fault-again's) are found in prefetch buffers in the case of 4K history buffers and 8 prefetch buffers.

Next, we performed a heavy-load multi-user test using a prototype prefetching system. We modified the 2.6MSD inode pager to prefetch pages based on fault history. Four users ran a combination of 4 types of programs at the same time. The four programs used in the test are:

matrix: 128 x 128 floating point matrix multiplication  
gcc: compile 19 sample programs with gcc  
locus: VLSI router  
espresso: integer benchmark (main operation is sort)

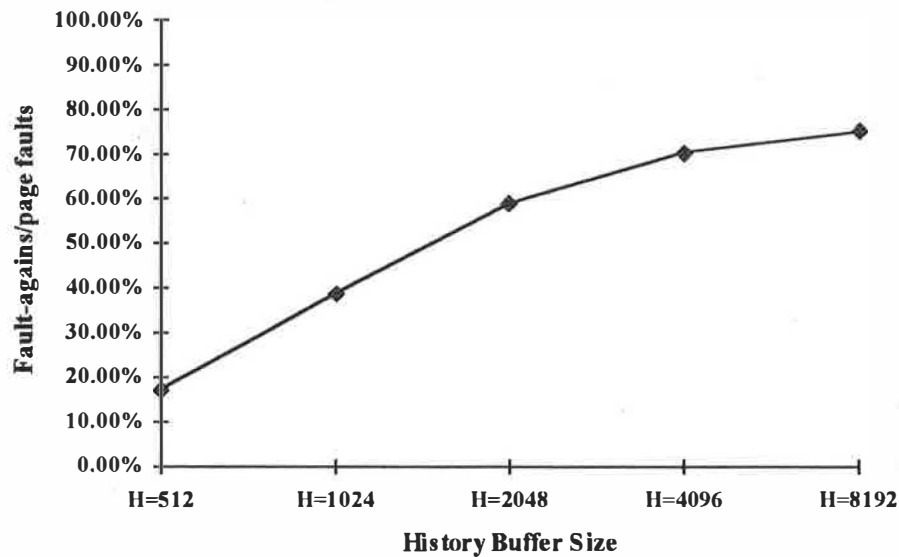


Figure 4-(a). Fault-again/Page-faults

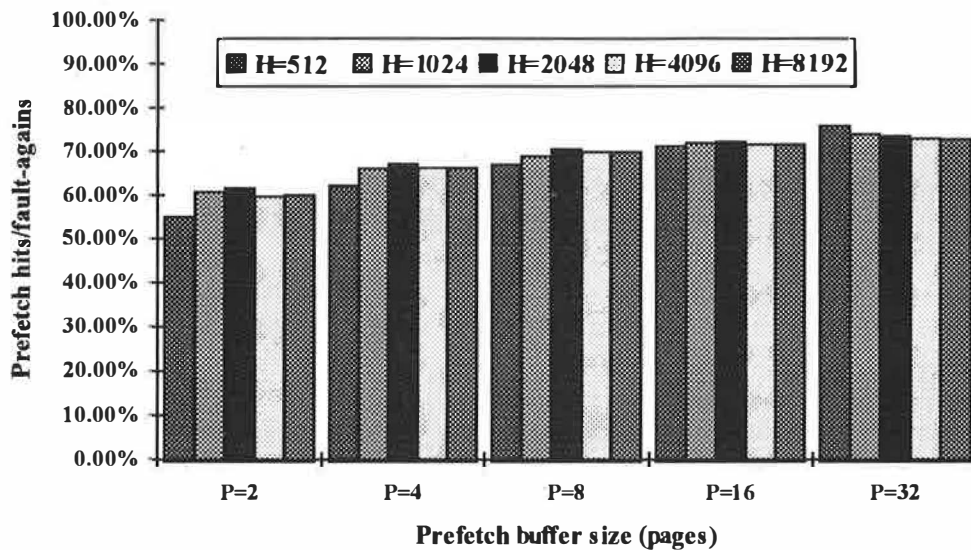


Figure 4-(b). Ratio of the prefetch hit counts/the number of pages faulted-again

Varying the number and the structure of history buffers and prefetch buffers, we have measured the time to execute workloads. In Figure 5 we see that the effective page-in delay time is reduced in most cases. The numbers of history/prefetch buffers tested are 1K - 8K entries and 4 - 8 pages respectively. Prefetch buffers are implemented as circular buffers because small amounts of the buffers are enough to contain prefetched pages. We compared two alternatives for history buffer structures, circular buffer and hashed buffers, because relatively large numbers of buffers are needed to more accurately predict the next pages to prefetch. Figures 5-(a), 5-(b), and 5-(c) in Figure 5 show the effects of history buffer size, prefetch buffer size, and history buffer structure respectively. Each measured time is the difference between elapsed time (in seconds) of the prefetching system and that of non-prefetching systems. The difference seems due to the reduction of page-in overheads. The total number of page faults was about 3,000, and the average 25 msec page-in delay has been measured. One second reduction of elapsed time means the effective reduction of 40 page faults.

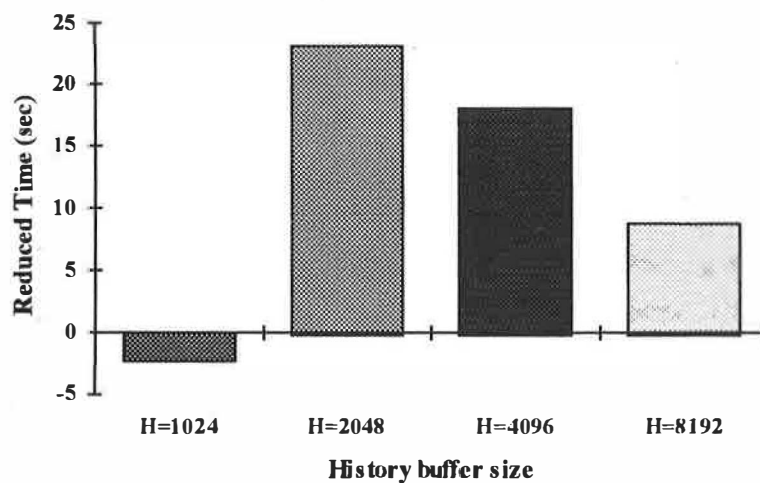


Figure 5-(a). History Buffer Size Effects (P=8 fixed)

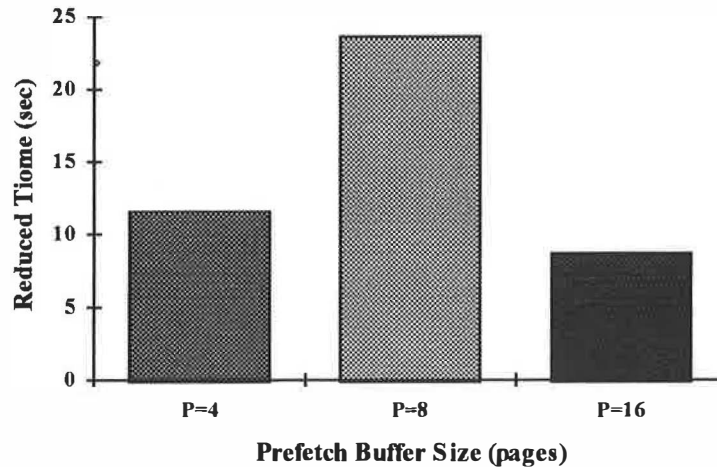


Figure 5-(b). Prefetch Buffer Size Effects (H=2048 fixed)

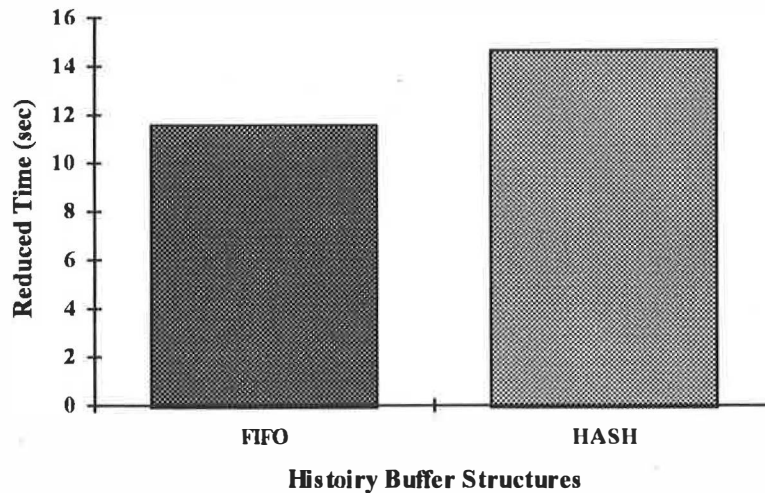


Figure 5-(c). History Buffer Structure Effects(H=2048, P=4 fixed)

In Figure 5-(a), the negative value observed in the case of 1K history buffers means that the prefetching overhead is greater than the gain. We observe that if the number of prefetch buffers is fixed to 8, the best result is obtained when 2K history buffers are used. There are several reasons for this. When we use small numbers of history buffers, the fault-again ratio becomes very low due to the limited capacity of histories stored, as shown in Figure 4. When the number of history buffers is large, the fault-again ratio does not increase as fast as the buffer count, also as shown in Figure 4. Also, more lookup time is needed and the number of free page frames decreases as the history buffer size increases. Figure 5-(b) shows that 8 prefetch buffers are best in the case when the history buffers size is fixed to 2K. Figure 5-(c) is the result of comparison between circular queue and hashed queue. Hashed queue outperforms the circular queue by 20%. In many test results, we observed a 10 to 20 second reduction in the case of the prefetch system that is equivalent to a reduction of 400 - 800 page faults. As a result, we get a 15% - 25% increase in performance under the multi-user test.

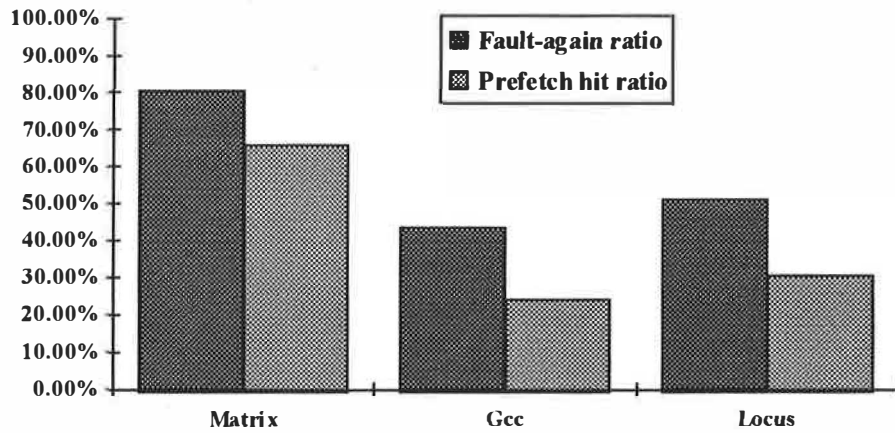


Figure 6. Application Dependency (H=1024, P=8 fixed)

Finally, we examine application dependency. In this case, we measured the fault-again ratio and the prefetch hit ratio for each of three applications, matrix, gcc, and locus, on the prototype prefetching system. Figure 6 is the result. As expected, the highest fault-again and prefetch hit ratios were measured for the matrix multiplication program. On the basis of this experimental result, our page prefetching system performs better for large repetitive applications, especially ones using matrices and/or vectors.

## 5. Future Work

Page prefetching based on fault history has been shown to yield good results in most cases, but prefetching does require more work to be done. More work to reduce prefetching overheads is needed. Page prefetching can be implemented at various points in the paging path. The kernel VM fault handler is one of those. The advantage of this approach is that we will be able to decide earlier in the paging path which pages are to be prefetched next, and that prefetched pages can be placed into the inactive page queue rather than into the separate buffers. We are trying to compare the pager implementation with the kernel VM implementation. Improving the accuracy of the prediction will be a further research area. We are also trying to implement page prefetching under Mach 3.0 and/or other systems such as SVR4 that have no separate inode pager mechanism.

## 6. Conclusions

We assumed memory access behavior is frequently repeated. Using this property, a page prefetching scheme based on fault history is presented. It may be used to enhance a paging system's performance by reducing unnecessary scheduling and idle time wasted in the demand-paging system. We showed the feasibility of the presented scheme with trace-driven simulation and prototype tests. Sequential prefetching is preferable in the initial phase of program load. Clustered paging enhances I/O bandwidth by combining adjacent pages in a single paging operation. Our prefetching scheme is well suited for more general situations, especially in large array applications.



## 7. Acknowledgements

I would like to thank Sangryul Min for suggesting the motivation for the prefetching. I also would like to thank program committee and Mike Hibler for reviewing the paper and proposing the further research area. Members of the System Software Research Lab. helped me to gather paging statistics and to prepare benchmark programs used in the the prototype system test.

## References

- [Black91] D. Black, et. al. "OSF/1 Virtual Memory Improvements," in Proc. USENIX Mach Symposium, Nov. 1991. pp. 87 - 103.
- [Callahan91] D. Callahan, et. al. "Software Prefetching," Proc. of 4th International Conference on Architectural Support for Programming Languages and Operating Systems, Apr. 1991, pp. 40.- 52.
- [Chow76] C. K. Chow, "Determination of cache's capacity and its matching storage hierarchy," IEEE Trans. on Computers, vol. c-25, Feb. 1976, pp. 157-164.
- [Denning80] P. J. Denning, "Working sets past and present," IEEE Trans. on Software Engineering, SE-6, 1, Jan. 1980, pp. 64 - 84.
- [Leffler89] S. J. Leffler et al. "The Design and Implementation of the 4.3BSD Unix Operating System". Addison-Wesley, Reading, MA, 1989.
- [Min92] S. Min, Personal communication, 1992.
- [Mogul91] J. C. Mogul and A. Borg, "The Effect of Context Switches on Cache Performance," Proc. of 4th International Conference on Architectural Support for Programming Languages and Operating Systems, Apr. 1991, pp. 75 - 84.
- [Rashid88] R. Rashid et al., "Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures," IEEE Trans. on Computers, vol. 37, no. 8, Aug. 1988, pp. 896 - 908.
- [Smith78] A. J. Smith, "Sequential Program Prefetching in Memory Hierarchies," Computer, Dec. 1978, pp. 7 - 21.
- [Subramanian91] I. Subramanian, "Managing Discardable Pages with an External Pager", Proceedings of USENIX Mach Symposium, Nov. 1991. pp. 77-85.
- [Trivedi76] K. S. Trivedi, "Prepaging and Applications to Array Algorithms," IEEE Trans. on Computers, vol. c-25, no 9, Sep. 1976. pp. 915 - 921.
- [Wang91] P. Wang and J. V. Sciver, "OSF/1 (1.0.1s+) System Memory Usage," OSF/1 Release 1.1 Engineering Note, Open Software Foundation, May, 1991.
- [Young87] M. Young at. al. "The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System", Proceedings of the Eleventh ACM Symposium on Operating Systems Principles, Vol. 21, No. 5, (Nov. 1987), pp. 63-77.



# Kernel Support for Recoverable-Persistent Virtual Memory<sup>1</sup>

Khien-Mien Chew<sup>2</sup>

*kenchew@cs.utexas.edu*

*Dept. of Computer Sciences, University of Texas, Austin, TX 78712-1188.*

A. Jyothy Reddy

*Hewlett-Packard Company, Cupertino, CA 95014.*

Theodore H. Romer<sup>3</sup>

*Dept. of Computer Science and Engineering, University of Washington, Seattle, WA 98195.*

Abraham Silberschatz<sup>4</sup>

*AT&T Bell Laboratories, 600 Mountain Ave., Murray Hill, NJ 07974.*

## Abstract

The buffering facilities typically provided by operating systems are not powerful enough to support the performance and consistency requirements of database systems. As a result, most database systems are structured as Buffer Pool Database (BPDB) systems, providing their own buffering facilities, with their own paging policies and recovery schemes. The emergence of operating systems with very large address spaces and flexible memory management makes Virtual Memory Database (VMDB) systems feasible. In such systems, the database is mapped into virtual memory and the buffering facilities of the underlying virtual memory system are used. VMDB systems do not experience many of the problems faced by BPDB systems. To support the consistency and recoverability requirements of VMDB systems, we have proposed that the virtual memory system be extended to support the Recoverable-Persistent Updates (RPU) model. This model is powerful and general enough to support a wide variety of policies for ensuring database recoverability. In this paper we discuss our approach to and progress in extending the Mach 3.0 kernel to provide direct support for this RPU model.

## 1. Introduction

A key component of modern database systems is the buffer system, which attempts to keep the most frequently accessed parts of the database in a buffer pool maintained in main memory. Although most operating systems provide buffering facilities, these are typically insufficient to support the performance and consistency requirements of a database system. Hence, most existing production database systems provide their own buffer management. A Buffer Pool Database (BPDB) system allocates a buffer pool within its own virtual address space, and is responsible for its own buffer management. There are a variety of performance problems and other disadvantages associated with this approach [1,2]. For example, the paging policies of a BPDB system and the underlying virtual memory system may interact poorly, resulting in greatly increased I/O costs due to double paging [9]. The BPDB approach may also be unsuitable for use in object-oriented database systems and computing environments with either very large main memories or limited swap storage [1].

An attractive alternative is to extend the virtual memory system to allow database systems to use the buffering facilities of the operating system, without compromising the integrity of the database [6]. Database systems can exploit the buffering facilities of the underlying virtual memory system by mapping

---

<sup>1</sup> This material is based in part upon work supported by the Texas Advanced Technology Program under Grant No. ATP-024, the National Science Foundation under Grant No. IRI-9106450, and grants from the IBM and Hewlett-Packard Corporations.

<sup>2</sup> Supported by a National Computer Board (Singapore) Postgraduate Scholarship.

<sup>3</sup> Supported by a NSF Graduate Research Fellowship and an ARCS Fellowship

<sup>4</sup> On leave from the University of Texas at Austin, where this work was performed.

the database into virtual memory [2,10]. We refer to this approach as the Virtual Memory Database (VMDB) approach. Compared to a BPDB system, a VMDB system has a substantially simplified and smaller buffering component. More importantly, a VMDB system does not suffer from many of the problems associated with BPDB systems because the virtual memory manager has direct access to the database and other system resources like main memory, and has accurate utilization information about them. The database is not duplicated in the swap space and updates are more likely to be reflected in the database since the virtual memory system is able to page the database directly to and from the mass storage on which the database resides. Unexpected and high I/O overhead associated with double paging is eliminated without having to statically allocate main memory since a single page replacement policy is used. Referencing a database object costs as little as referencing a transient target [11]. Virtual memory hardware can be exploited to speed up address translation and access control. Thus, the VMDB approach presents opportunities for improved performance, ease of programming and efficient use of resources in the memory hierarchy. Some examples of VMDB systems are TABS [16], Camelot [2], ObjectStore [11] and CPR [18].

A VMDB system, like a traditional database system, must be able to deal with failures and recovery. Updates modify database records in volatile virtual memory. System crashes can occur at any time, resulting in the loss of such data. As a result, the updates need to be *made persistent* on nonvolatile storage if they are to persist beyond the lifetimes of the programs that performed them. Also, updates may need to be undone or redone in order to restore the database to a consistent state [12]. If a database can be recovered to a consistent state in the event of such failures, then it is said to preserve the *recoverable-persistence* property and a database state that has this property is referred to as a recoverable-persistent state. Database systems use database recovery techniques that ensure that the database is always in a recoverable-persistent state and, if necessary, can transform such a state to an appropriately consistent state after recovery processing following system failure [13,14,15].

Essentially, recoverable-persistence is preserved by propagating updates to the database on nonvolatile secondary storage and imposing propagation ordering constraints on them. Since updates are propagated at the granularity level of pages, page propagation must be performed in such a way that these propagation constraints are maintained. Unfortunately, no such support for recovery and consistency exists in current day virtual memory systems. Page replacement is performed independent of these propagation constraints: for example, commonly encountered replacement policies seek to keep the most frequently used pages in memory. Hence, a VMDB system cannot simply be implemented on top of existing virtual memory systems. Existing VMDB systems solve this recovery problem by employing solutions that compromise some key benefits of the VMDB approach, and that support only either a particular recovery technique or class of recovery strategies. Thus, existing approaches are inadequate.

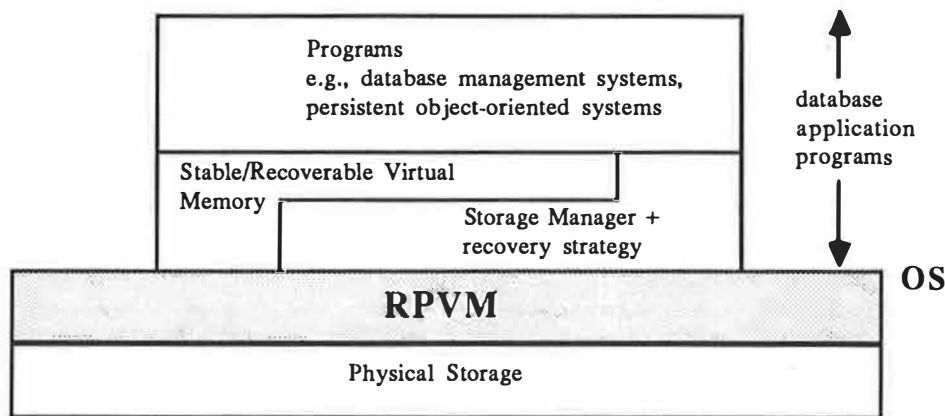


Figure 1. Our Recovery Support Approach.

Our work explores the implications of a different approach to providing operating system support for recoverable-persistence in VMDB systems. Our approach is based on supporting a novel Recoverable-

Persistent Updates (RPU) model [1] in the operating system kernel. The small set of simple primitives provided by the RPU model can be used to support a wide variety of recovery schemes, rather than only a specific strategy or class of strategies. The model allows greater flexibility with regards to the ordering of data record updates and recovery-related updates (e.g., logging). Thus, the RPU model is simple, general and flexible, and is highly suitable to being implemented in an operating system kernel. We refer to virtual memory that is extended to support the RPU model as Recoverable-Persistent Virtual Memory (RPVM).

VMDB systems built on top of RPVM can make use of both its file memory-mapping and recovery support facilities to realize all the benefits of the VMDB approach, without compromising their consistency requirements. Such VMDB systems can choose to implement their own recovery schemes using the primitives provided or to use those strategies provided by the RPVM system, if any is available. In the former case, the VMDB system is responsible for recovery processing following failures, while in the latter case, the underlying system uses some recovery scheme to restore the database to a consistent state on behalf of the VMDB system. In addition, such VMDB systems can specify that recoverable-persistence be maintained for only a selected part of the mapped database. For the remainder of the database, recoverable-persistence is not maintained and regular file memory-mapping semantics apply. The propagation constraints that are central to ensuring recoverable-persistence can be specified either explicitly via system calls or implicitly in the database. The relationship between the RPVM, database system and the rest of the computer system is depicted in Figure 1.

In this paper, we describe our initial experiences in modifying the virtual memory system of the Mach 3.0 kernel [19] to directly support this model. This involved adding a small RPVM module to the kernel, some small changes to the virtual memory page-in and page-out paths to communicate with this module, as well as the addition of a set of new system calls for manipulating the primitives provided by the RPVM module. From the point of view of the virtual memory system, the RPVM module looks very much like an external pager. This similarity will become clear as we describe the implementation. However, there are significant differences between the two and there exist compelling reasons for embedding the RPVM module in the kernel.

## 2. Existing Approaches

One approach, adopted by the TABS prototype [16], is to modify the virtual memory system to always consult a user-level recovery manager or process before it propagates any modified database page, say P. The recovery manager ensures that those pages that must be propagated before P have been propagated before sending a message to the kernel, informing it that P may be propagated. The page P is written back to the database only after the virtual memory system has received this message from the recovery manager. This approach incurs much messaging overhead. The TABS system implements both value and operation logging, together with the Write-Ahead Log protocol, to realize database recoverability.

Another example of a VMDB system is a database system built using Camelot [2], which takes advantage of Mach's external pagers [8] to provide recoverable virtual memory. In Mach, external pagers are responsible for writing the pages of memory-mapped files back to nonvolatile storage. In Camelot, the Disk Managers are responsible for propagating database pages in an order that does not violate propagation constraints. They are implemented as external pagers. However, since external pagers are user-level tasks with virtual memory, and pages that cannot be immediately propagated need to be buffered, Camelot's Disk Managers are essentially BPDB systems. Thus, Camelot can still suffer from double paging and the other problems that plague BPDB systems, albeit to a somewhat smaller degree.

Camelot also requires that a specific pin-modify-log protocol be followed by all database systems that modify the database. This protocol requires a number of messages to be exchanged between the database system and Camelot for every database update. Again, only value and operation logging is supported and the Camelot system restores the database to a consistent state after a failure before the database can be accessed. As a result of these characteristics, embedding Camelot in an operating system kernel will not yield a sufficiently general and flexible recoverable virtual memory system.

A commercial product that maps the database into virtual memory is ObjectStore [11]. However, ObjectStore does not use file memory-mapping. Instead, when a database page is accessed for the first time, a memory fault occurs and is serviced by ObjectStore, which requests the required page from a database server. Once a page has been cached in the database system's virtual memory address space, the cost of

referencing data on that page is identical to that of a VMDB system. Also, modified pages can only be propagated back to the database server by ObjectStore. To the virtual memory system, database pages are no different from regular transient virtual memory. As a result, the buffer system architecture of a database system built using ObjectStore is very similar to that of a BPDB system with a buffer pool that is as large as the database. Hence, such a database system cannot fully realize all the benefits of the VMDB approach. Also, ObjectStore's database server uses a specific page-level log-based recovery strategy which is transparent to the database system.

In CPR [18], hardware support is provided for recoverable virtual memory. Unfortunately, the support is not based on a general recovery support model but implements a specific and limited recovery strategy. Recovery based only physical logging of entire database pages is supported.

Our work differs from the previous work in two ways: the recovery support we propose is flexible and general, and our prototype implements such support at the operating system kernel level, potentially realizing all the benefits of the VMDB approach.

### 3. The Model

Since the focus of this paper is on the implementation of kernel support for the RPU model in the Mach kernel, we will not attempt to justify that model in detail here [1]. Rather, we will only discuss the salient features of the model. We will also describe the additional semantics that are required for virtual memory to support this model.

#### 3.1 Recoverable-Persistent Updates (RPU)

In the RPU model, all data accessed by a database system forms the database, including recovery-related data that is generated and maintained for the purposes of facilitating recovery after a crash or when the effects of updates need to be undone or redone. For example, log records generated by logging schemes are recovery data. Updates may affect any data in the database and since they are performed in volatile main memory, they must eventually be propagated to nonvolatile storage in order to be persistent. The model treats all data (and their updates) uniformly, whether they are recovery-related or not.

The database is memory-mapped into the virtual address space of the database system in the same fashion as memory-mapped files. The definition of consistency is application-specific and a database state is consistent as long as it is meaningful or useful to the application concerned. For example, a definition for consistency for most database systems is transaction-based atomicity and durability. We say that a persistent database state is *recoverable-persistent* if the state is either consistent, or it can be transformed into one by a *recovery procedure*. A database is recoverable-persistent if its state is always recoverable-persistent.

A key characteristic of recovery schemes for transaction management in database systems is the existence of ordering constraints on the propagation of updates to the database in nonvolatile storage. For example, in the Write-Ahead-Log protocol, the updates that modify data records must not be propagated before their corresponding log entries have been propagated. Propagation ordering constraints are also central to shadow paging, multi-level recovery [17] and other recovery schemes [13,14]. As a result of this observation, we conclude that in order for a database to be recoverable-persistent, its updates must be made persistent in some partial order. This partial order is application and recovery scheme specific, and in the RPU model, it is specified in the form of *flush rules*, which can be of two types:

- **Atomic Flush (AF).** Such a rule specifies that a set of updates be propagated atomically. We refer to such a set of updates as a *Set of Atomic Persistence (SAP)*.
- **Flush Before (FB).** Such a rule specifies a propagation precedence relationship between two sets of updates. A FB rule is of the form  $A \leq B$ , where  $A$  and  $B$  are sets of updates, and it specifies that  $A$  must either be propagated before  $B$  is propagated or be propagated atomically with  $B$ . If a crash should occur and  $B$  is persistent, then  $A$  must also be persistent.

A *flush policy* is a set of flush rules and these flush rules are the central feature of our RPU model. It can be shown that these two types of flush rules are sufficiently powerful to express all propagation constraints,

given our assumptions, and that the RPU model is sufficiently general to support the realization of a wide variety of recovery strategies [1].

### 3.2 Recoverable-Persistent Virtual Memory

In most virtual memory systems, the unit of data transfer between main memory and nonvolatile storage is the *page*. The virtual memory address space is made up of non-overlapping pages. Existing virtual memory systems neither support the notion of high-level updates nor guarantee the atomic propagation of updates that span multiple pages. Thus, if existing virtual memory is not to be drastically altered, direct support for flush rules over the propagation of updates in the RPU model is not possible.

Instead, we propose that the RPU model be supported in virtual memory by extending the kernel to support the following two types of page-level constraints on the propagation of virtual memory pages that are mapped to a database:

- **Flush-locks.** When taken, these prevent pages that are flush-locked from being propagated to the database. They are provided essentially to maintain intra-page consistency, since the virtual memory system is unable to detect when individual high-level updates begin and end in general. Unlike pinning, they are intended to be held for only the duration of individual high-level updates; and flush-locked pages may be paged to the swap space.
- **Page-Flush Before (P-FB) Rules.** Such a rule specifies a propagation precedence relationship between two pages. A P-FB rule is of the form  $A \leq_p B$  where  $A$  and  $B$  are pages, and is used to specify that whenever pages  $A$  and  $B$  are dirty, page  $A$  must either be propagated before  $B$  is propagated or be atomically propagated with page  $B$ . Also, if page  $A$  is flush-locked, page  $B$  cannot be propagated. We can use P-FB rules to effectively specify the same propagation constraints as those that can be specified by the AF and FB flush rules of the RPU model. We refer to page  $A$  as the *potentially constraining* page and page  $B$  the *potentially constrained* page of the P-FB rule  $A \leq_p B$ . Page  $A$  is a *constraining* page if it is a potentially constraining page and it either has been modified or is flush-locked. Page  $B$  is a *constrained* page if it has been modified and page  $A$  is a constraining page.

The set of flush-locks and P-FB rules pertaining to a database constitute the database's *page-flush policy*. When a dirty page of a database is selected for replacement by the virtual memory system's replacement policy, but cannot be propagated according to the database's page-flush policy, it will be paged out to the swap space. Otherwise, it is written back to the database on nonvolatile storage. Conceivably, an errant program could cause the swap space to be filled with such pages, but we will not be concerned with this issue in this paper.

We refer to a virtual memory system that is extended in the above manner to support the RPU model as *Recoverable-Persistent Virtual Memory* (RPVM). The resultant RPVM maintains a page-flush policy for each database and guarantees that each database is recoverable-persistent by ensuring that database pages are paged out in a manner that does not violate any relevant propagation constraint. Page-flush policies are specified by user-level database and application systems.

The flush-locks taken and the P-FB page-flush rules specified depend on the specific recovery strategies being supported, as well as the granularity of updates and the order in which they are performed. The order in which these propagation constraints need to be added to a page-flush policy also depends on the relative ordering of the updates they constrain. We have developed a protocol, called the Flush-Before (FB) protocol [1], that specifies the flush-locks and page-flush rules that must be added to a page-flush policy and the order in which they are to be added relative to the updates under a variety of different conditions. For example, consider the situation in which an update to page  $A$  must be propagated either before or with the propagation of an update to page  $B$ , and page  $B$ 's update is performed before page  $A$ 's update. After page  $B$ 's update is completed, page  $B$  must remain flush-locked until the page-flush rule  $A \leq_p B$  has been added and page  $A$  has been flush-locked or updated. Page  $A$ 's flush-lock can be removed as soon as its updates have been completed. If all database systems using RPVM adhere to this FB protocol, recoverable-persistence is ensured.

## 4. RPVM User Interface

Here we discuss how a database system or user-level task can make use of the facilities provided by RPVM to specify propagation constraints. Although this description of the user interface is tailored for use in the Mach environment, the interface can be easily supported by other operating systems provided they support virtual memory and file memory-mapping.

Before a database can be accessed using the VMDB approach, it must first be memory-mapped into the virtual memory address space of a task (for purposes of brevity, we will use the term *task* to include database systems). This is accomplished by the `vm_map()` call (or `mmap()` call in most UNIX operating systems). The page-flush policy for that database may be specified by that task or another related task (for example, the data manager task of a database system). In our prototype, a propagation constraint can only be added if the database pages to be constrained are currently memory-mapped into some task's virtual memory address space. The address location of a memory-mapped database page within a task's virtual address space is used to identify or name the database page.

More than one task can memory-map a database (external memory object semantics apply) and, likewise, more than one task may update the page-flush policy associated with the database. Unless explicitly removed, page-flush policies persist in the virtual memory system beyond the lifetimes of the tasks that update them, until all Mach kernel references to the database have been deallocated. No additional operations are required before a task can unmap a database or terminate. The existence of flush-locks beyond the lifetime of the tasks that took them may indicate an error condition but the handling of such errors is beyond the scope of this paper. The RPVM model does allow the virtual memory or some underlying system to provide atomic propagation and recovery of a selected set of database pages. This support can be activated whenever P-FB rules that form a cycle are encountered in a page-flush policy and all the pages involved are dirty. In our initial prototype, however, we are primarily concerned with handling P-FB rules that do not form cycles. Thus, neither atomic propagation of a set of pages nor recovery processing after a failure is implemented in the prototype.

### 4.1 Page-Flush Policy

Tasks can add and remove flush-locks and page-flush rules to and from page-flush policies. In our prototype, tasks do so explicitly via new system calls. While propagation constraints must be added in a timely fashion, they need not be removed. However, flush-locks and P-FB rules that are no longer necessary may unnecessarily constraint page propagation and add additional computational and I/O overhead to paging activities. Thus, such constraints should be removed.

In our prototype RPVM system, we have extended the kernel interface to include the following new RPVM interface system calls:

```
kern_return_t  rpvm_flush_lock(
    target_task      :    vm_task_t;
    begin_address    :    vm_address_t;
    num_bytes        :    vm_size_t);
```

The `rpvm_flush_lock` system call flush-locks a range of pages that belong to a database.

```
kern_return_t  rpvm_flush_unlock(
    target_task      :    vm_task_t;
    begin_address    :    vm_address_t;
    num_bytes        :    vm_size_t);
```

The `rpvm_flush_unlock` system call removes the flush-locks of a range of pages that belong to a database.

In our prototype system, we implemented *memoryless* flush-locks. These flush-locks do not retain information about the identity of the tasks that added them or the number of such tasks. Such information



may be required in systems where multiple tasks concurrently update and flush-lock shared database pages. Memoryless flush-locks would be inadequate in such situations. Non-memoryless or *memoryful* flush-locks could also be useful in handling and controlling errant task behavior. A memoryful flush-lock would keep a count of the number of times it has been added less the number of times it has been removed or unlocked. Such a flush-lock will be removed only if a `rpvm_flush_unlock` call results in this count being decremented to zero. However, since it is preferable to maintain the information associated with memoryful flush-locks in user-space, only memoryless flush-locks are supported directly.

```
kern_return_t    rpvm_add_rule(
    target_task    :    vm_task_t;
    rule           :    user_rule);    /* struct[2] of vm_offset_t */
```

The `rpvm_add_rule` system call adds a P-FB rule to the page-flush policy of a database. The P-FB rule is specified in *rule*. The first element of *rule* identifies the potentially constraining page (i.e., page A of the P-FB rule,  $A \leq B$ ). The second element of *rule* identifies the page whose propagation is potentially constrained (i.e. page B). The P-FB rule is added to the page-flush policy associated with the database to which page B belongs.

In our current implementation, both the potentially constraining and constrained pages of a P-FB rule are assumed to be from the same database. The identity of the *target\_task* and the number of times a particular rule has been added are also not recorded. As in the case of memoryful flush-locks, such information may be needed in some environments and should be maintained in user-space.

```
kern_return_t    rpvm_remove_rule(
    target_task    :    vm_task_t;
    rule           :    user_rule);
```

The `rpvm_remove_rule` system call removes a P-FB rule from a page-flush policy of a database. The P-FB rule is specified in *rule*. The semantics of this system call are similar to those of `rpvm_add_rule` in all other aspects. The rule is removed from the policy of the database containing the potentially constrained page. It is the responsibility of the calling task to ascertain that removing the rule will not compromise recoverable-persistence.

## 4.2 Explicit Page Propagation

Tasks may need to explicitly request that a particular set of modified database pages be propagated, and moreover, may require that they be notified when such propagation requests have been successfully completed. For example, in many log-based recovery strategies, a transaction is considered to be committed if and only if certain log records have been propagated to stable storage. Thus, the RPVM interface also provides the following system call to support transaction semantics:

```
kern_return_t    rpvm_propagate_request(
    target_task    :    vm_task_t;
    begin_address  :    vm_address_t;
    size           :    vm_size_t;
    should_flush   :    boolean_t;
    should_rp_propagate :    boolean_t;
    reply_port     :    mach_port_t);
```

The `rpvm_propagate_request` system call allows a task to request that all the pages in a range of pages that belong to a database and that have been modified be propagated. The range of pages is specified by the parameters *begin\_address* and *size*. The pages in the range are also flushed from the kernel's cache if *should\_flush* is set to TRUE. If *should\_rp\_propagate* is set to TRUE, propagation proceeds in an order that does not violate all relevant page-flush policies. Directly or indirectly constraining pages that are not in the

range specified are also propagated. Due to propagation constraints that exist, some pages in the specified range may not be propagated. Such situations are indicated by a suitable return code.

This system call preserves recoverable-persistence only if the *should\_rp\_propagate* argument is TRUE and the pages that are propagated are not modified until the system call returns. A similar system call that does not require the second condition to hold in order to preserve recoverable-persistence may be useful and can be implemented but is not currently provided in our prototype.

The *rpvm\_propagate\_request* system call is an asynchronous call. The *reply\_port* is provided to realize synchronous page propagation. A message from the external pager backing the database concerned is sent to the *reply\_port* to signal the successful propagation of all propagatable pages in the specified range.

### 4.3 Other Related Extensions

In the previous sections, we described the basic system calls that make up the RPVM user interface. Extensions to this interface can allow propagation constraints to be specified and page-flush policies to be updated more concisely, but they do not increase the fundamental power of the model on which RPVM is based. Such extensions may be important in client-server systems where the above RPVM calls are made by remote client tasks.

For example, page-flush rules could be specified over sets of pages rather than over individual pages. Another extension would allow page-flush rules to be specified over pages belonging to different databases. In the current RPVM implementation, if propagation constraints exist between pages of different files on nonvolatile storage, the external pager has to handle and map the files in such a way that they appear to the kernel as a single database (or external memory object). This abstraction simplifies the RPVM and is consistent with our design goal of providing minimal but sufficient kernel support.

Another way to reduce the cost of processing page-flush rules and the cost of explicitly removing rules is to support automatic page-flush rule removal in the RPVM. Such a facility would allow tasks to terminate sooner than if they had to explicitly remove the rules that they have added. We will not describe the details of such a automatic rule removal scheme and its use in this paper. The *rpvm\_add\_rule* call would have to be extended to support such a facility.

Before a user task performs an update on a page, it must first flush-lock the page, unless the update can be performed atomically. After the update has been performed, the flush-lock can be removed. Thus, we can expect flush-locks on database pages to be taken very frequently and for extremely short intervals of time. Using the *rpvm\_flush\_lock* and *rpvm\_flush\_unlock* system calls will likely impose too high an overhead. A solution to this problem is to implement *flush-lock tables* that are shared between user tasks and the kernel [5]. To flush-lock a particular database page, a task simply turns on a bit in a flush-lock table that corresponds to its flush-lock. To flush-unlock, the bit is turned off. The kernel need only read the corresponding bit entry in the flush-lock table to determine the flush-lock status of any page. The kernel will never block when it does so, since setting a flush-lock bit can be performed in a single XOR machine instruction. Such support can also be used to implement fast memoryful flush-locks in user space.

### 4.4 External Pager Interface

RPVM requires a simple extension to the external pager-kernel interface. In the existing Mach 3.0 external pager-kernel interface, the external pager provides a *memory\_object\_data\_write* procedure [19] that is commonly used to write a set of pages to their backing memory object on nonvolatile storage. The writes are neither guaranteed to be performed immediately nor in the order that they received by the external pager. For example, the UX server's pager attempts to optimize disk I/O by buffering and/or reordering file write requests. Thus, recoverable-persistence may not be maintained if RPVM makes use of such a procedure to propagate database pages. Moreover, the existing external pager-kernel interface does not allow the external pager to inform the kernel of successful propagations. Support for synchronous page propagation will be impossible.

Therefore, in addition to the kernel modifications described above, we also need to extend the external pager-kernel interface and modify external pagers to support the following:

- **Partially Ordered Page Propagation.** The external pager must guarantee that pages will be written to nonvolatile storage in either some fixed partial order or according to some partial order specified by the kernel. This will ensure recoverable-persistence as long as the partial order used does not violate page propagation constraints. In our prototype, we modified the inode pager of the UX server to provide a `rpvm_memory_object_data_write` procedure which guarantees that writes to nonvolatile storage are performed in the order in which write requests are received from the kernel. Although this ordering is overly strict, it allows recoverable persistence to be maintained. This procedure is similar to the `memory_object_data_write` procedure in all other respects.
- **Page Propagation Notification.** This allows the kernel or a user task to request that a notification be sent when a particular page propagation has been successfully performed. Such a facility enables the synchronous `rpvm_propagation_request` call to be supported by the kernel. In our prototype, the UX server's new `rpvm_sync_memory_object_data_write` procedure sends a message to a given port provided by the kernel when the write request has been performed successfully. It is similar to the `rpvm_memory_object_data_write` procedure in all other respects.

In the case of operating systems that do not have user-level external pagers, such support will have to be provided within their kernels and device drivers.

## 5. Implementation

We chose the Mach 3.0 kernel as the platform for our experiment. It can be argued that, rather than modifying the kernel, it would have been more appropriate and simpler to add RPVM support to an external pager (for instance, by augmenting the semantics of the Unix file system); however, we chose instead to make our changes directly in the kernel. We defer presenting the arguments in support of this choice until the next section, after we have described our implementation.

To implement RPVM in the kernel, two separate implementation issues had to be addressed. The first is the modifications to the virtual memory system to enable it to recognize the page-flush policies and to perform memory management and paging in a manner consistent with the policies. The second is the new kernel data structures that are needed to store and enable access to these page-flush policies. Essentially, a small RPVM module that provides data structures for storing page-flush policies and some functions for updating and accessing them were added to the kernel. A small number of changes were also made to the virtual memory page-in and page-out paths to communicate with this module.

### 5.1 Changes To Mach Virtual Memory

Virtual memory in Mach is managed in terms of memory objects [7]. A memory object whose paging to and from its backing store is handled by an external pager is referred to as a *permanent* memory object. For instance, the files and databases that are memory-mapped by user tasks are permanent memory objects. In RPVM, we have extended such memory objects with an optional field to denote a special class of memory objects called *RPVM memory objects*. If the RPVM field is present, the pages belonging to the object are subject to the propagation constraints specified in the page-flush policy associated with it. A RPVM memory object may be shared by multiple tasks and its page-flush policy updated by any or all of them. However, coordination among the different tasks in user space with regards to updating the page-flush policy may be needed, for example to prevent deadlock or cyclically related page-flush rules [1]. As implementors of RPVM, we just have to be certain that user-level system calls provided by RPVM do not deadlock the kernel.

The interface between the existing virtual memory system and the RPVM module is straightforward. When a dirty page in a RPVM memory object is selected for replacement, the pageout daemon calls a function in the RPVM module to determine whether the page can be propagated safely (i.e., sent to the external pager). If not, that is it is constrained, the page is sent to the default memory manager to be written to the swap space. Doing so allows memory to be freed while maintaining recoverable persistence since the database is not updated and as far as the external pager is concerned, the page is still cached in

main memory. Similarly, when a page fault occurs on a page in a RPVM memory object, the page fault code calls a function to determine whether the page should be retrieved from the external pager or the swap space. Memory management and the paging operations involving pages from all other regular permanent memory objects, and temporary memory objects (that are backed by the default memory manager) remain unchanged. In the rest of this section, we are concerned only with the pages of RPVM memory objects.

One key function of the RPVM module is to determine the propagatability of a database page that has been selected for replacement. If the page is constrained, it cannot be propagated. Since a user can specify arbitrary rules constraining the propagation of a page, and since the RPVM data structures reside in the kernel's virtual memory, the RPVM module can take an arbitrary length of time and memory to determine the propagatability of a page. If the pageout daemon synchronously calls this RPVM module function, in the worst case, all available free memory may be consumed before the call returns, causing the pageout daemon to block [8]. Since the pageout daemon is the only source of free pages, it cannot be blocked waiting for free pages. Hence, the daemon cannot check the propagatability of a page synchronously in this way. There are a few possible ways to address this propagatability checking problem:

- **Asynchronous Checking.** In this approach, the pageout daemon calls the function asynchronously - the pageout daemon sends an asynchronous message to the RPVM module, requesting it to determine the propagatability of a page selected for replacement. If the RPVM module is too slow to respond or physical memory is in short supply, the page may be paged to the swap space as if it was constrained, thus freeing the physical page. A variant of this technique that does not alter the replacement policy is for modified database pages in the inactive queue to be checked by a daemon before they are even selected for replacement. If such a page is determined to be propagatable, then it is flagged as so. If it is subsequently moved to the active queue and modified again, then the flag is reset. Whenever the pageout daemon selects a flagged and modified database page for replacement, it is propagated. Otherwise, the modified page is considered to be constrained and sent to the swap space.
- **Kernelized Pager.** In this approach, the pageout daemon also calls the function asynchronously but the RPVM module behaves much like an external pager. Rather than calling a boolean function directly to determine the propagatability status of the page, the pageout daemon sends an asynchronous message to the RPVM module, requesting it to dispose of the dirty page. If the RPVM module is too slow, or physical memory is in short supply, the page may become eligible for replacement again, and may be double-paged to the swap space, freeing the physical page. Similarly, when a page fault occurs on a page in an RPVM memory object, the page fault handler sends a message to the RPVM module requesting it to retrieve the page. Implemented this way, the RPVM module is essentially an external pager resident within the kernel, except that it may explicitly send pages to the swap space. For the purposes of our initial experiments, we have taken this approach and the asynchronous messages are provisionally implemented as synchronous procedure calls.
- **Bounded Checking.** In this approach the checking function is called synchronously. However, to prevent the pageout daemon from blocking, a limit is placed on the number of propagation rules that are checked. An upper bound on the number of pages accessed by the RPVM module function can then be computed. As long as the sum of this upper bound and the original allowance made for the pageout daemon is less than the size of the reserved memory pool [8] of the daemon, blocking will not occur. If the threshold is reached, the function returns negatively, that is the pageout daemon should handle the page as if it is constrained. If most pages are not constrained by many rules, such an approach is practical.

## 5.2 Ordered Page Propagation.

Page replacement is only one case in which unconstrained RPVM memory object pages are sent to their respective pagers. The other cases are when a RPVM memory object is terminated, and when a user task requests explicitly that a set of pages belonging to a RPVM memory object be propagated (by using the `rpvm_propagate_request` call). In these cases the system must ensure that pages are propagated to the database in the correct order, by constructing a graph of the dependencies between all concerned dirty pages.

The current implementation of the `rpvm_propagate_request` call assumes that all potentially constraining pages that are required to be propagated are not updated from the time the call is made until it returns. If not, recoverable-persistence may not be preserved. To understand why this is the case, consider the implementation of the `rpvm_propagation_request` call. Let the dirty pages A and B be related by the rule  $A \leq_p B$ , and consider the actions taken when a user task requests that page B be propagated. Page A is propagated before page B in order to maintain database recoverable-persistence. This is achieved by making two `rpvm_memory_object_data_write` calls, one for page A followed by another for page B.

Now, consider the case where pages A and B are updated before the system call has been completed. The updates may occur after the kernel thread yields the processor as a result of insufficient memory resources while attempting to propagate page A using a `rpvm_memory_object_data_write` call. During the time this thread is inactive, another user task may update pages A and B, and also require that the same propagation constraint hold between these updates. Under the current implementation, when memory resources become available, the waiting kernel thread completes the `rpvm_memory_object_data_write` of page A, propagating the old version of page A. It then proceeds to propagate page B which contains new updates. Thus, the new updates on page B are propagated before those of page A, resulting in a loss of database recoverable-persistence. In a multiprocessor computer, this scenario can also occur even if the kernel thread does not yield its processor.

A naive potential solution to the above problem, that does not involve disabling page access by user tasks, is to have the kernel thread reexamine the state and propagatability of page A before proceeding with the propagation of pages A and B. It will discover the new version of page A and may even discover that additional pages may constrain either pages A or B. Such a reexamination of the page-flush policy and the states of potentially constraining pages may be repeated indefinitely in the presence of continuous updates and large graphs of dependent pages, leading to a situation where the actual propagation of page B never takes place. Thus, such a technique is not desirable.

Another solution is for the kernel thread to acquire a copy of page B (if B is not flush-locked) before it attempts to check the status of page A and propagate it if necessary. Thus, even if page B is updated again, the version of page P that is eventually propagated by this thread will be the older and correct version, and recoverable-persistence will have been maintained.

This technique allows page A to be updated after the kernel thread has determined it to be clean or has propagated it, without compromising recoverable-persistence. This is because no propagation constraint can exist between these later updates to page A and any update reflected in the copy of B (or B') that was acquired prior to the processing of page A. If a P-FB constraint exists between the new version of page A and B' (that is,  $A \leq_p B'$ ), and the FB protocol (see Section 3.2) was adhered to, then B' would have been flush-locked while page A was being checked and propagated. However, page B was not flush-locked when a copy of it was acquired. Thus, propagating B' following further updates to A will not violate recoverable-persistence. Note that the order in which the copy of B is acquired and page A is propagated is important. If the copy of B is acquired after page A has been propagated, recoverable-persistence may not be preserved.

Similarly, additional rules and flush-locks that are added after the copy of B has been acquired need not be considered by the kernel thread propagating page B. For a large graph of dependent pages, a tree of such copies will have to be made but if copy-on-write is available the space and page copying overhead is only paid when the original pages are actually updated. Such a technique can be used to provide a variant of the `rpvm_propagate_request` call which allows concurrent database page updates, rule additions and page propagation to occur without compromising recoverable-persistence.

### 5.3 RPVM Data Structures

The data structures in the initial version of the RPVM module support a small set of operations on the pages of RPVM memory objects, namely:

- **User Interface Operations:**
  - Adding and removing P-FB rules;
  - Adding and removing flush-locks; and
  - Ordered propagation of a set of pages.

- **Kernel Operations:**

- Determining the set of P-FB rules that defines the set of pages that potentially constrain a given page;
  - Determining whether a page is flush-locked;
  - Determining the pager from which to request a page
  - Recording the identity of modified pages that have been paged to the swap space; and
  - Ordered propagation of a set of pages.

For the initial prototype, we chose to implement relatively straightforward data structures, deferring tuning of the data structures until we had some performance numbers in hand. There are three main data structures associated with each RPVM memory object in this initial implementation:

- **SwapHash Table.** A table to record which pages belonging to the RPVM memory object have been diverted to the swap space. This is implemented as a simple hash table with chaining. An entry is inserted by the pageout daemon when it writes a dirty page to the swap space. This table is consulted by the page fault handler when a fault occurs for a page belonging to the RPVM memory object and when such pages that are not in main memory have to be propagated. An entry is removed when such a page is fetched from the swap space.
- **Flush-lock List and Tables.** An ordered linear list to record the ranges of pages that are explicitly prohibited from being propagated. To support fast flush-locking and removal of such locks, we have also implemented streamlined, shared memory-based flush-lock tables [5] which allow direct user access to flush-locks without kernel intervention. These structures are consulted by the page propagation procedure and pageout daemon before any page belonging to a RPVM memory object is propagated.
- **RuleTable.** A RuleTable contains the P-FB rule-based propagation constraints defined over the pages of the RPVM memory object. Conceptually, this is a two-dimensional array indexed by page offset within the object. It is in fact implemented by two hash tables. One hash table is indexed by offset, and its elements are linked lists of pages potentially constraining the page at the given offset (for example for page at offset B, the list of all pages A such that  $A \leq B$ ). The other hash table is also indexed by offset, but its elements are linked lists of pages that can potentially be constrained by the page at the given offset (for example for page at offset A, the list of all pages B such that  $A \leq B$ ). These structures are also involved in the determine propagatability step of the page propagation procedure and pageout daemon.

The above data structures are allocated only when needed and deallocated when empty, and are allocated on a per-object basis. In fact, a memory object is a RPVM memory object if and only if either a Flush-lock or RuleTable structure has been allocated for it. Thus, these structures are not allocated for non-RPVM memory objects. Also, if database pages are not paged to the swap space, the SwapHash Table is not allocated.

Many of the data structures were implemented in C++, building on the GNU class libraries [3]. The hope is that by using C++, replacing the data structures will prove relatively painless. Integrating C++ code into the kernel was straightforward, requiring only a slight modification to the memory allocation routines.

## 6 Kernel vs External Pager

It is apparent from the previous section that the RPVM module operates very much like an external pager. Indeed, in principle the RPU model can be implemented outside the kernel in a specialized external pager. However, we chose to base our work on a kernel-based RPVM system because of the limitations imposed by the external pager-kernel interface and the opportunities for improved kernel performance presented by direct kernel support.

## 6.1 Limitations of an External Pager-based RPVM

One difficulty with implementing the RPVM mechanisms in an external pager is that the external pager may have outdated information on which pages are clean or dirty. Consider the case where the RPVM module is implemented in an external pager and it has received a page, say B, from the kernel. Let there be a P-FB rule that indicates that page A potentially constrains page B. Page A has been provided to the kernel. Even though it may have been clean when it was last seen by the external pager, page A could have been modified in the kernel's cache. Thus, before it can propagate page B, the external pager needs to determine the modification state of page A. This information must be determined through an exchange of messages between it and the kernel involving the use of the functions `memory_object_lock_request()` on page A, and `memory_object_lock_completed()` or `memory_object_data_write()`. If page A is clean, the external pager may proceed to propagate it, provided there are no other constraints.

If page A is dirty and can be propagated, the external pager propagates it before proceeding with the propagation of B. However, if updates affecting page A and the above message exchange can occur concurrently, that is page A is allowed to be flush-locked and updated after the `memory_object_lock_request` on A is called and before a copy of page A from the kernel is received, then the copy of page A that is acquired by the pager may not be consistent. This problem is avoided if updates are blocked while the flush-lock status of page A is being determined and a copy of page A is acquired. Hence this problem does not arise in a RPVM that is implemented within a uniprocessor kernel whose threads are not preemptible. However, the problem can still arise in a kernel-based RPVM running on a multiprocessor computer system.

The external pager may also have outdated information concerning which pages are in main memory. This may result in the external pager making poor choices about the order in which to write pages back to disk. This in turn may result in more disk accesses than necessary. For example, consider the case where a set of pages {A, B, C, D} can be propagated in any order and where only page A is in the swap space. Since the external pager is not aware of page A's location, it may choose to propagate page A first. The paging in of page A may lead to page B being moved to the swap space. Next it chooses to propagate page B, resulting in page C being swapped out and so on. Instead of incurring only a single I/O operation to fetch page A from the swap space, the propagation of the set of four pages incurs a total of seven swap space I/O operations.

An external pager may need to buffer in virtual memory those database pages that cannot be propagated. As a result, it can also experience increased I/O costs associated with the double paging anomaly. This can result in a significant increase in the number of I/O operations incurred. Since avoiding this anomaly is one of the key reasons for using the VMDB approach, using external pagers in this way to realize RPVM systems is unsatisfactory. If the RPVM module is implemented within the kernel, then double paging can still occur. However, double paging can be avoided if the RPVM module can avoid sending the page being considered for propagation to either the swap space or the database when the page has already been paged out to the swap space. An external pager would not be able to accurately detect such a condition given the existing pager interface and that it can be preempted.

We note that while it is possible to construct pathological examples, experiments will ultimately be required to determine whether these performance costs are significant in practice.

## 6.2 Placing RPVM in the Kernel

A kernel implementation of the RPVM module is superior to an external pager-based implementation for three key reasons. First, in a uniprocessor environment, the non-preemptibility of kernel threads allows the modification state of pages to be determined simply and with little overhead. Second, accurate information about the state of the machine, memory use, and the location and state of database pages can be accessed with little overhead. Third, constrained pages that are selected for replacement can be paged to the swap space explicitly. As a result, most of the problems mentioned above that apply to an external pager-based implementation do not apply to a kernel-based implementation.

In addition, by integrating the RPVM module with the kernel, the potential for improved kernel operations exists. For example, page replacement policies could be optimized with information maintained by the RPVM module. For example, if pages A and B have both been modified and the rule  $A \leq_p B$  holds,

the replacement policy may select page A together with page B, if both are on the inactive queue. If Mach provided support for user-level replacement policies [4], this would be a lesser concern. Prefetching of database pages from the swap space based on the semantic information found in P-FB relationships could also improve virtual memory performance. We also hope to show that the writing of modified pages back to their backing stores can be accomplished more efficiently with this approach.

## 7. Performance Evaluation

It is our hypothesis that kernel support for RPVM does not introduce significant additional overhead to existing virtual memory operations, and that it can also improve the overall performance of database programs. In this section, we provide evidence in support of the first part of our hypothesis.

Implementing RPVM in the kernel requires modifications that primarily extend the virtual memory page-in and page-out path lengths. Among virtual memory operations, the page-in and page-out operations are among the most frequently invoked. It follows that the computational costs incurred by these paging operations can have a significant effect on virtual memory and program performance. Hence, our initial evaluation efforts were centered on the impact of the RPVM modifications on these paging costs. Both analytical and initial experimental data suggest that a kernel-based RPVM does not in general significantly impact virtual memory performance.

### 7.1 Page-in Analysis

Virtual memory page-in operations, invoked as a result of page faults, fetch pages from their pagers, and these pages are typically read from nonvolatile storage. In RPVM, the following additional steps may be performed by the page-in operation:

- **Ascertain RPVM object.** Ascertain if the page belongs to a RPVM memory object. If so, the next two operations may be necessary. Otherwise, the original page-in path is taken.
- **Locate page.** Ascertain if the required page is in the swap space. If not, the traditional page-in path is taken and the page is requested from its external pager.
- **Swap space page-in.** If the page is in the swap space, fetch it in from the default memory manager.

In kernel-based RPVM, all page-in operations have to perform the *ascertain RPVM object* step that can involve only a single read of a RPVM field, followed by a compare instruction. This negligible overhead is the only additional direct cost to page-in operations that handle pages belonging to non-RPVM memory objects.

In the case of memory page-ins involving RPVM memory objects, the *locate page* step is also performed. The cost of this additional step is the cost of checking whether the required page has been paged to the swap space. Recall that in our prototype this information is stored in a hash table. If none of the pages of the memory object has been modified or there has been no page replacement involving its modified and constrained pages, then the hash table will be empty and the cost of this step should be small.

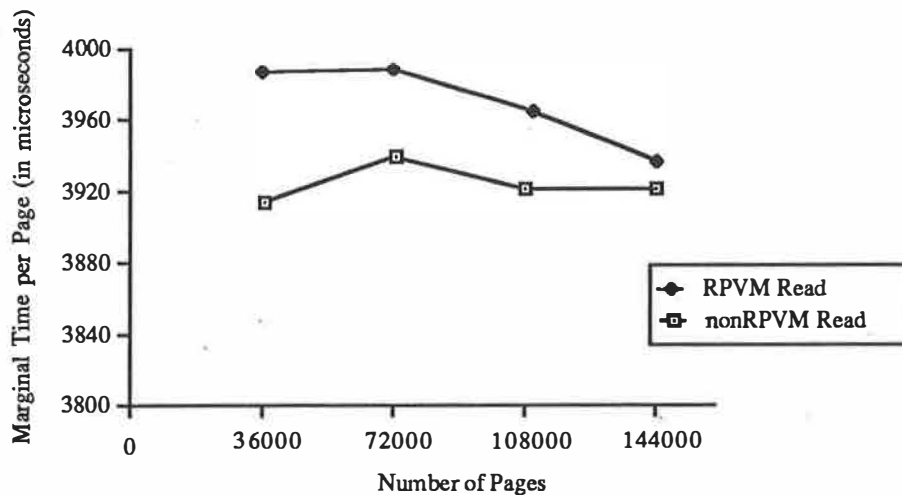
If the required page has been paged to the swap space, the *swap space page-in* step is performed -- the page is requested from the default memory manager instead of its external pager. The page is in the swap space only if the required page had previously been modified and selected for replacement. Preliminary results show that for this step to be as efficient as traditional page-in operations that fetch pages solely from external pagers, the default memory manager must support prefetch.

### 7.2 Page-in Performance

We conducted our experiments on a prototype RPVM operating system running on a Sun 3/60 workstation with 16 Mbytes of main memory. The prototype RPVM system was derived from the Mach 3.0 microkernel (version MK75) with a user-level Unix-server (version UX38). The system uses a virtual memory page size of 8 Kbytes. We used an external pager (the modified Unix-server pager) that was extended to satisfy kernel data requests and writes without performing real I/O or paging on behalf of the



page-in and page-out operations being measured. This allowed us to use wallclock time to represent the computational time taken by the kernel to perform the paging operations. We report the measured times in microseconds per paging operation.



This graph shows the marginal times for each additional page-in operation for pages from non-RPVM and RPVM memory objects, after page replacement has begun. The time taken by a program to read a given number of pages of the memory object was measured. The page-in times for the RPVM object are for the case when zero pages of the object have been paged to the swap space.

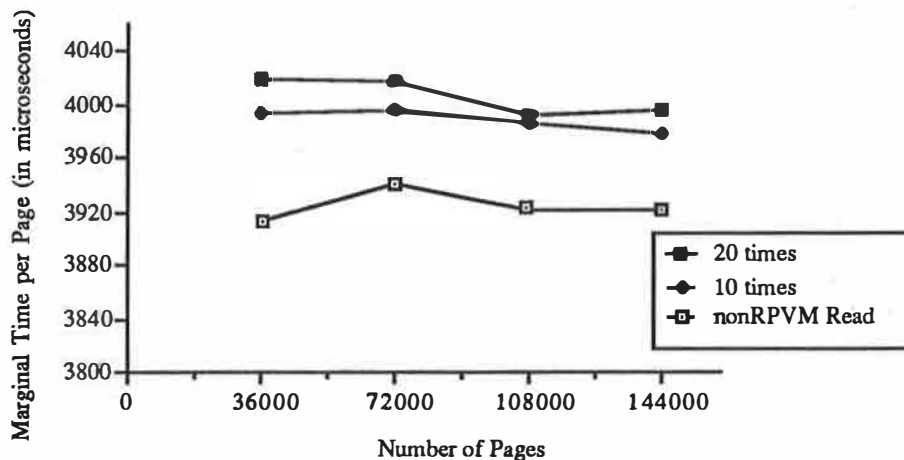
Figure 2: Measured Performance of Basic RPVM Page-in Operations

To measure the overhead incurred by a page-in operation, we used a simple program that mapped a file, representing a database and managed by a user-level external pager, into the program's address space. The program sequentially reads one byte of each page in a range of pages belonging to the mapped database. The time taken by the program to perform these reads for a number of pages or sample size represents the time taken by the kernel to perform the same number of page-in operations. To measure the cost associated with page-in operations involving pages from a RPVM memory object, the same program was used except that memory-mapped database was made a RPVM memory object by adding some suitable propagation constraint on its pages. For each experiment, the times taken for different sample sizes were measured. We used the marginal time taken to perform a page-in operation to represent the cost of such an operation as it eliminates the effects of page-ins at the beginning of each run that did not involve page replacement. The marginal time for a sample size is computed by dividing the difference in the time measured for this sample size and the time for the next smaller sample size by the difference in sample sizes.

To ascertain the additional cost of the *locate page* step when the swap space does not contain pages of a RPVM memory object, we measured the average time taken by a program to read and page-in a page belonging to the object from its external pager. We compared this to the case where the memory object read is a non-RPVM object. The difference between the two measured times represents the cost of the *locate page* step. The measurements from this experiment are shown in Figure 2. The observed average difference is 45 microseconds per page read. Thus, the additional cost incurred by the *locate page* step is 1.15% of the read page-fault cost of a non-RPVM object in this scenario.

The swap hash table is not empty if some RPVM pages are in the swap space. However, if an appropriate swap hash function is used, then the additional cost of the *locate page* step will remain small. To verify this, we created a scenario where a large number of constrained pages from a RPVM memory object were replaced from main memory and paged into the swap space. Then we measured the average time taken by a program to read and page-in the remaining pages of the RPVM object from its external pager. We found that when the number of swapped pages is less than or equal to the number of buckets in the swap hash table, the additional overhead incurred by the *locate page* step was negligible when compared with the case when no pages were swapped. When the number of swapped pages increases to 10 times the

number of buckets, the *locate page* step overhead is only 18.5 microseconds more than if the swap hash table is empty or 1.6% of the cost of a non-RPVM object page-in. When the number of pages increases to 20 times the number of hash buckets, the overhead increases to only 36 microseconds more than if the swap hash table is empty or 2% of the cost of a non-RPVM object page-in. The measurements from this experiment are shown in Figure 3. From this data, we conclude that implementing RPVM in the kernel has very little impact on the performance of page-in operations.



This graph shows the marginal times for each additional page-in operation for pages from a RPVM memory object, after page replacement has begun and for the case where 1001 pages of the object are in the swap space. We show the times for the cases where the number of pages in the swap space is 10 and 20 times the number of buckets in the swap hash table. The time taken by a program to read a given number of pages of the memory object was measured. The pages in the swap space are neither accessed by the program, nor were they paged there as a result of its reads.

Figure 3: Measured Performance of RPVM Page-in with Pages in the Swap Space

### 7.3 Page-out/Replacement Analysis

Page-out operations send pages to their respective pagers, and these are typically written to nonvolatile storage. In the Mach kernel, this operation is invoked by the pageout daemon when modified pages resident in main memory are selected for replacement. In RPVM the page-out operation may perform the following additional steps:

- **Ascertain RPVM object.** Ascertain if the selected page belongs to a RPVM memory object. If the page is not from a RPVM object, the original pageout path is taken. Otherwise the next two steps may also be performed.
- **Determine Propagatability.** Determine if the page can be propagated. This involves retrieving all relevant flush rules and flush-locks that specify propagation constraints on that page and processing all rules that apply.
- **Swap space Page-out.** If the page cannot be propagated, page it to the swap space and record its location. Otherwise, the original pageout path is taken.

In kernel-based RPVM, all page-out operations perform the additional *ascertain RPVM object* step. As in the case of page-in, this negligible overhead is the only additional direct cost page-out operations that handle pages belonging to non-RPVM memory objects.

The *determine propagatability* step is performed if and only if the selected page has been modified and belongs to a RPVM memory object. We refer to a page-flush rule, say  $A \leq_p B$ , as a potentially constraining rule of page B. A rule is a constraining rule of page B if it is a potentially constraining rule of

page B, and page A is either flush-locked or has been modified. We use the term *policy size* to refer to the number of different propagation constraints (flush-locks and rules) that constitute a policy.

Depending on the data structures used, the cost of retrieving all potentially constraining rules of a selected page may be sensitive to the total number of rules present in its page-flush policy. In our prototype system, rules are indexed via a hash function. If an appropriate hash function is used and there are not too many pages that have rules specified, then this cost should be independent of the policy size. The same holds for accessing flush-locks. Thus, the cost of the *determine propagatability* step is unlikely to be affected by the policy size but is probably highly dependent on the existence and number of potentially constraining rules. For each RPVM page being paged-out, one of three different cost scenarios applies.

First, if no propagation constraints apply, then the selected page can be propagated and the cost of this step is the cost of determining that no flush-locks and potentially constraining rules exists for the selected page. This cost is independent of the policy size if the policy is not very large.

Second, if the selected page has potentially constraining rules but is not constrained, then it can also be propagated. In this case, the cost of the *determine propagatability* step is directly proportional to the number of such potentially constraining rules. For each potentially constraining rule that is found, the state of the corresponding potentially constraining page has to be obtained. Since none of the rules constrain the selected page's propagation, all such rules have to be retrieved and processed in this way, unless we short-circuit the process by having a threshold. Thus, this processing is probably the greatest potential source of overhead.

Third, if the selected page has a constraining rule, and thus cannot be propagated, the cost of the *determine propagatability* step depends directly on the number of potentially constraining but unconstraining rules that are processed before a constraining rule is encountered.

To facilitate the estimation of the cost incurred by the *determine propagatability* step in the case where there are potentially constraining rules, we define the *flush-rule ratio* of a page to be the ratio of the total number of constraining rules to the total number of potentially constraining rules of the page. As the ratio approaches 0, the average cost of this step is increasingly dependent on the number of potentially constraining rules. On the other hand, as it approaches 1, the average cost of this step becomes increasingly independent of the number of such rules and approaches the cost of processing a single potentially constraining page-flush rule.

The *swap space page-out* step is only performed for pages that cannot be propagated and which have been selected for replacement. Before such a selected page can be paged to the swap space, the identity of the page and related information are inserted into the swap hash table so that the page can be located when it is subsequently required in main memory. The cost of this update operation should be comparable to the cost of the *locate page* step of the RPVM page-in operation. Except for this additional cost, the swap space pageout cost incurred should be no higher than if the selected page was sent to its external pager.

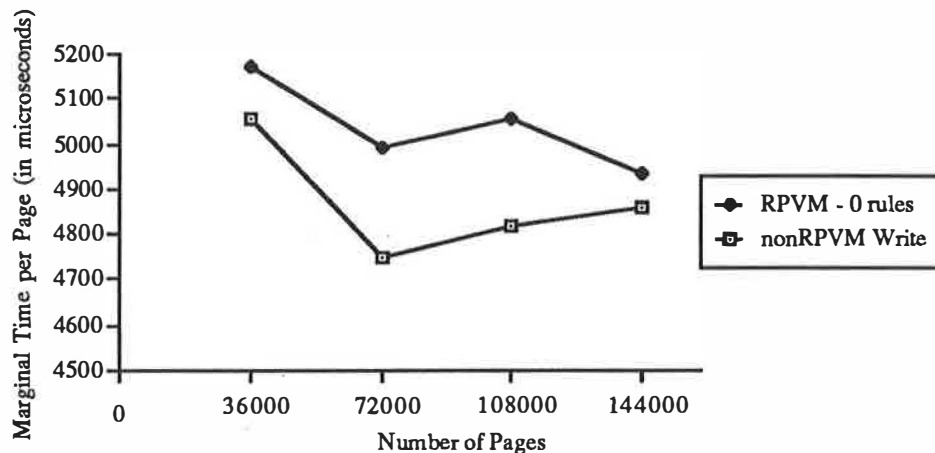
## 7.4 Page-out Performance

The same kind of measurements taken for page-in operations were also taken for page-out operations. The performance of a program that sequentially writes one byte of each mapped database page was measured. These modified pages are paged-out back to their external pager when page replacement occurs.

We measured the time taken by a program to write and the kernel to page-out modified RPVM pages when no propagation constraints apply. The average additional overhead incurred by the *determine propagatability* step was 169 microseconds or 3.4% of the average cost (4869.5 microseconds) of handling a single page fault. The measurements are shown in Figure 4. Since page-outs are invoked to make memory resources available for page fault processing, we compare the additional page-out cost to the average computational cost of processing a single page fault.

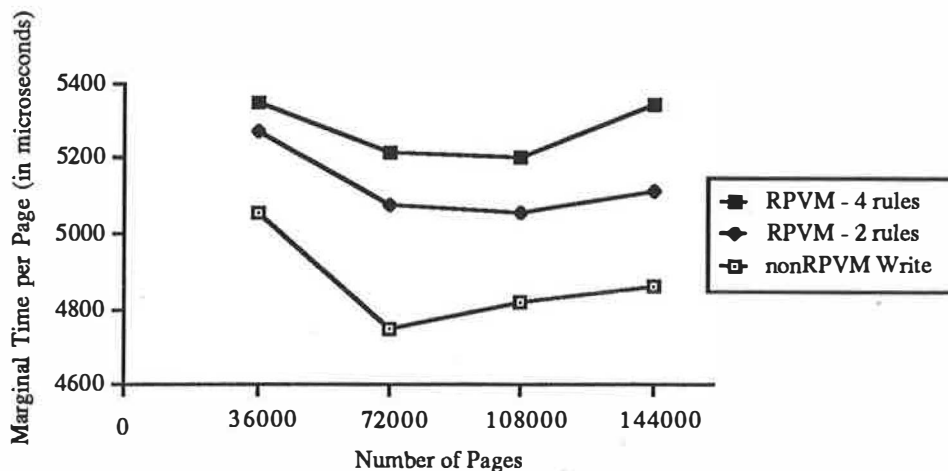
Our experiments indicate that if a small number of potentially constraining page-flush rules exists for the pages being paged-out, then the additional cost incurred by the *determine propagatability* step is not significant. In the situation where the flush-rule ratio is 0, that is there are no constraining rules, and there are 2 potentially constraining rules for each database page that is replaced, the average page fault increased by 262 microseconds or 5.4%. With 4 potentially constraining rules per page, the page-fault cost increased by 410 microseconds or 8.4%. These measurements are shown in Figure 5. The average marginal cost per additional potentially constraining rule is 74 microseconds or 1.5% of the page fault cost of a non-RPVM

object. Thus, when the flush-rule ratio is close to 1 or when the average number of potentially constraining rules per RPVM page is not too large, we do not expect support for RPVM to significantly impact page-out and page fault performance.



This graph shows the marginal times for satisfying each additional page fault for non-RPVM and RPVM memory objects, after page replacement has begun. The time taken by a program to write a given number of pages of the memory object was measured. The marginal page fault time consists of the marginal page-in cost as well as the amortized marginal cost of page-out operations. The page-fault times for the RPVM object are for the case where the pages replaced have zero potentially constraining rules and none of the pages of the object have been paged to the swap space.

Figure 4: Measured Performance of RPVM Write Operations



This graph shows the marginal times for satisfying each additional page-fault involving pages from a RPVM memory object, after page replacement has begun and for the cases where there are a given number of potentially constraining rules for each RPVM page. The time taken by a program to write a given number of pages of the memory object was measured. The marginal page-fault time consists of the marginal page-in cost as well as the amortized marginal cost of page-out operations. We show the page-fault times for the cases where each replaced page belonging to the RPVM object has 2 and 4 potentially constraining rules, and compare them to the page faults times of non-RPVM memory objects.

Figure 5: Measured Performance of Write Operations with Potentially Constraining Page-Flush Rules.

## 7.5 RPVM Performance

From our analysis and experiments, it is clear that negligible additional computational overhead is experienced by existing applications that do not use the new RPVM primitives.

Paging operations involving pages of RPVM objects, however, do experience non-negligible additional overhead. The time taken to perform a page-in operation for a page of a RPVM memory object is dependent on the number of pages from that object that have been paged to the swap space. This in turn is dependent on the amount of main memory allocated to database pages and the percentage of database pages in main memory that have been modified and are not propagatable. Only if these two factors combine to result in a very large number of database pages being paged to the swap space and a significant increase in *locate page* step costs will any significant degradation in average page-in performance be observed.

The time taken to perform a page-out operation involving a page of a RPVM memory object is dependent on the size and nature of the page-flush policy corresponding to the memory object. The additional overhead incurred is particularly sensitive to the number of potentially constraining rules of the page, especially if the page's flush-rule ratio is close to zero. Although in principle, there can be as many such rules as there are pages in an address space or database, in practice this number is likely to be small. Moreover, a limit can be placed on the number of rules that are processed. If this threshold is reached, the page can be treated as if it is constrained.

## 8. Conclusions

Our initial work indicates that it is feasible to add direct and general support for Recoverable-Persistent Virtual Memory based on the RPU model to the Mach kernel. We were able to do so by building upon existing kernel data structures and functions in a modular way and by making small changes to the virtual memory page-in and page-out paths. The RPVM user-interface introduces only a few new system calls and abstractions. We believe that other operating systems that support memory-mapped files can similarly be extended to realize RPVM. The RPVM system adequately supports the consistency and recoverability requirements of VMDB systems while potentially allowing all the benefits of the VMDB approach to be realized.

We have examined the impact of the RPVM module on the performance of simple programs that use virtual memory and the new RPVM primitives, and found that our modifications do not impose significant virtual memory overhead. Our results indicate that the retrieval and processing of potentially constraining rules can potentially be a source of significant overhead. Further studies are needed to ascertain whether commonly used recovery techniques result in page-flush policies with flush-rule ratios close to zero and a large average number of potentially constraining rules per page and if so, how this problem can be addressed. Other future work includes comparing the performance of this system with a similar system implemented as an external pager, and studying how the kernel's page replacement policy and database system performance can be further improved by exploiting information present in the page-flush policies of the RPVM module.

## References

- [1] Chew, K. and Silberschatz, A. Toward Operating System Support for Recoverable-Persistent Main Memory Database Systems. Tech. Rept. TR-92-05, Dept. of Computer Sciences, University of Texas at Austin, February 1992.
- [2] Eppinger, J.L. *Virtual Memory Management for Transaction Processing Systems*, Ph.D. dissertation, Carnegie Mellon University, February 1989.
- [3] Lea, D. *User's Guide to the GNU C++ Library*, Free Software Foundation, Inc., (1992).
- [4] McNamee, D. and Armstrong, K. Extending the Mach External Pager Interface to Allow User-level Page Replacement Policies. Tech. Rept. 90-09-05, Dept. of Computer Science and Engineering, University of Washington, September 1990.

- [5] Reddy, A.J. *Implementing Recovery Support for Virtual Memory Databases in Mach 3.0*, Masters thesis, University of Texas at Austin, August 1992.
- [6] Stonebraker, M. Operating System Support for Database Management. *CACM* 24, 7 (July 1981), 412-418.
- [7] Tevanian, A. *Architecture-Independent Virtual Memory Management for Parallel and Distributed Environments: The Mach Approach*, Ph.D. dissertation, Carnegie Mellon University, December 1987.
- [8] Young, M. *Exporting a User Interface to Memory Management from a Communication-Oriented Operating System*, Ph.D. dissertation, Carnegie Mellon University, November 1989.
- [9] Goldberg, R.P. and Hassinger, R. The double paging anomaly. *Proc. AFIPS National Computer Conf.* 43(May 1974), 195-199.
- [10] Shekita, E.J. *High-Performance Implementation Techniques for Next-Generation Database Systems*, Ph.D. dissertation, University of Wisconsin - Madison, May 1991.
- [11] Lamb, C., Landis, G., Orenstein, J., and Weinreb, D. The ObjectStore Database System. *CACM* 34, 10 (October 1991), 50-63.
- [12] Korth, H. and Silberschatz, A. *Database System Concepts*, McGraw-Hill Inc, New York, NY, McGraw-Hill Computer Science Series, 2nd edition (1991).
- [13] Haerder, T. and Reuter, A. Principles of Transaction-Oriented Database Recovery. *Computing Surveys* 15, 4 (December 1983), 287-317.
- [14] Verhofstad, J.S.M. Recovery Techniques for Database Systems. *Computing Surveys* 10, 2 (June 1978), 168-195.
- [15] Mohan, C., Haderle, D., Lindsay, B., Pirahesh, H., and Schwarz, P. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. *ACM Transaction on Database Systems* 17, 1 (March 1992), 94-162.
- [16] Eppinger, J.L. and Spector, A.Z. Virtual Memory Management for Recoverable Objects in the TABS Prototype. Tech. Rept. CMU-CS-85-163, Dept. of Computer Science, Carnegie Mellon University, December 1985.
- [17] Weikum, G., Hasse, C., Broessler, P., and Muth, P. Multi-Level Recovery. In *Proc. of ACM PODS*, 1990, pp. 109-123.
- [18] Chang, A. and Mergen, M.F. 801 Storage: Architecture and Programming. *ACM Transaction on Computer Systems* 6, 1 (February 1988), 28-50.
- [19] Loepere, K. *Mach Kernel Principles*. Open Software Foundation and Carnegie Mellon University, Open Software Foundation Mach 3 Series, Revision 2.2, (July 1992).

# Real Memory Mach <sup>(1)</sup>

Philippe Bernadat<sup>(2)</sup>  
David L. Black<sup>(3)</sup>

## Abstract

Mach was designed with the assumption that the underlying hardware includes a page-based memory management unit, enabling the use of a large, sparse virtual address space. Therefore the Mach micro-kernel in its current state cannot be used on segmented machines such as Cray vector supercomputers or real memory machines such as transputers. This paper describes the results of initial work to adapt the Mach micro-kernel to such architectures, including the architectural changes and results obtained from a prototype system.

## 1 Introduction

Mach's virtual memory system is based on the abstraction of a memory object[7]. A memory object represents a single source of memory (e.g., an executable file), and task address spaces are assembled by mapping portions of memory objects into the address space. For example, a traditional process consists of the text and data portions of one object plus additional objects for uninitialized (bss) memory and the stack. Mach's communication (IPC) system allows portions of memory objects to be included in messages for efficient transmission of large amounts of data[6]. Communication and interaction among tasks is also facilitated by the ability to share memory objects among tasks. The implementation of Mach's virtual memory (VM) system, including memory objects, makes extensive use of page-based functionality to support techniques such as lazy evaluation and copy on write. As a consequence, the existing Mach micro-kernel cannot be easily adapted to machines that do not support page-based memory management; these include machines that only support segments, and machines with no memory management support whatsoever.

Memory management hardware that supports pages allows virtual addresses to be mapped to arbitrary physical addresses on a page by page basis. Protection (read/write) and mapping validity can also be set on a page by page basis; a memory access for which a valid mapping does not exist or that exceeds the protections allowed by its mapping causes the processor to take a trap. The operating system is then responsible for providing a valid translation with sufficient protection, or signalling an exception to the application. Operating system implementations of virtual memory functionality (including Mach's) depend on the resulting ability to modify virtual to physical memory mappings and protections without the application's knowledge.

Page based memory management support is not universally available. Although virtually all workstations and minicomputers support this functionality, other systems do not. For some high performance processors, the impact of translation logic on hardware cycle time may be unacceptable (e.g., Cray systems[8]). Another reason is that the impact of translation on processor complexity may be unacceptable (e.g., transputers[3]). These hardware driven design decisions are often complemented by system usage patterns that place less demand on memory management functionality (e.g., dedicated use, swapping instead of paging); the resulting system software can be closely tied to its hardware base. We refer to such hardware as *real memory systems*.

---

1. This work was supported in part by a grant from Cray Research, Inc.

2. Philippe Bernadat, OSF Research Institute, Grenoble, France, bernadat@gr.osf.org

3. David L. Black, OSF Research Institute, Cambridge, MA, USA, dlb@osf.org

Software that does not depend on large sparse virtual address spaces may not need sophisticated memory management support. Much of the Unix interface does not require the existence of virtual memory. Implementations are another matter; both OSF/1[4] and OSF/1 MK (the Mach 3 based single server version of OSF/1[1][2]) are designed for platforms that support page-based memory management. OSF/1 MK should be more amenable to execution on hardware without virtual memory support because its server uses Mach kernel interface (as opposed to accessing internal kernel routines) and has little dependence on the existence of virtual memory. Our results bear this out; only a few minor configuration changes were needed to adapt the OSF/1 MK server to our prototype implementation of Mach for segmented hardware without virtual memory support.

This paper is organized as follows:

- Segmented memory model
- Objectives and Approach
- Mach VM & IPC architecture description
- Micro kernel architectural changes
- Server modifications
- Prototype Status
- Performance
- Future work

## 2 Segmented Memory Model

An abstract segment model forms the basis of our work to adapt Mach's memory management implementation for execution on hardware that lacks virtual memory support. Depending on the specific architecture, these segments can be of fixed or variable size, and may or may not have alignment constraints. We use the term *virtual segment* to refer to a segment in a task's address space, and the term *physical segment* to refer to a contiguous region of physical memory that may be accessed via a virtual segment. These notions of segment replace the corresponding notions of virtual and physical pages in communication between the machine independent and machine dependent (pmap) portions of the memory management code. The pmap module for the resulting real memory system is responsible for mapping, unmapping, and protecting segments instead of individual pages. The correspondence with hardware features is obvious for segmented hardware, but this model is also applicable to hardware that has no memory management support due to its (desirable) property of managing physical memory in contiguous chunks.

The model delineates virtual segments by their segment number and length; physical segments are correspondingly identified by a base physical address and length. We assume that the segment number of a virtual segment is implicit in addresses supplied to Mach's kernel VM primitives (e.g., `vm_allocate()`). Use of the most significant bits to encode the segment number is a common technique. This model is also applicable to hardware that specifies the segment number in a different fashion for actual memory accesses; in this case it is necessary to define a composite address for use in kernel primitives that includes the segment number (encoding in the high bits is one possible technique). Examples of hardware for which this is necessary include that in which the segment number is implicit in the type of access (e.g., instruction vs. data) or is explicitly specified in the instruction (e.g., by selection of segment register).

We have made minimal conservative assumptions about the capabilities of the hardware for this work. No assumption is made about whether accesses to invalid data can be continued or restarted (they are fatal on some segmented systems). As a result, we have had to remove all lazy evaluation from Mach's memory management code. Hardware that can restart or continue such accesses may be able to take advantage of techniques such as load on reference, or copy on write, on a segment basis. Similarly, we make no assumptions about the hardware's ability to enforce read, write and execute protections. In fact, we take advantage of this to limit segment usage. For example, on hardware that cannot enforce protections, all protections are equivalent and a new segment need never be created for protection reasons.



An overall goal of this work has been to consider virtual segments as a scarce resource, and manage them accordingly. For example, Cray Research machines have exactly two segments, a text segment and a data segment. Since these systems implement stacks as software abstractions in the data segment, this is equivalent to three segments on a machine that supports stacks in hardware. Our prototype implementation has not been optimized for architectures where a task could map an arbitrary number of segments with arbitrary protections.

### 3 Objectives and Approach

The goal of this work has been to modify Mach's memory management system to execute on real memory hardware with minimal impact on the existing kernel interface. In order to address the fundamental architecture and design issues without getting distracted by the details involved in a complete port to a new system, we have built a prototype system that emulates segments using a conventional memory management unit. The prototype implementation was intended to address the memory management architecture issues and kernel interface changes. Additional work on the prototype would be necessary to define formal segment layers and machine dependent interfaces. The architecture described in this paper occasionally corresponds to the ideal goal rather than the actual prototype; we have been careful to make this distinction clear when it arises.

The required kernel interface modifications are minor, and usually consist of subsetting to cope with the limited capabilities of real memory hardware. For example, external memory managers are still supported (e.g., to obtain contents of executed and mapped files), but the kernel will return an error if a manager attempts to change the protection of an individual page. Another example of a hardware limit is that every explicitly mapped object (e.g., a mapped file) consumes one virtual segment; attempts to use more segments than the hardware supports are also rejected by the kernel.

Our prototype kernel has a number of limitations. We have not addressed swapping of segments or management of swap space. We have also made no attempt to run our modified Mach kernel on actual segmented hardware. This kernel also does not support multiprocessors, although there are no fundamental design obstacles in extending it to do so. This study has only considered variable sized segments without alignment constraints. The ideal design would split segment management into machine independent and machine dependent portions to avoid the appearance of segment constraints in machine independent code.

The prototype has reached the point where a port to segmented hardware would be reasonable to undertake (i.e., there are no fundamental architectural or design issues standing in the way of such a port). We have been pleasantly surprised by the small amount of work required to run the OSF/1 MK server on the prototype kernel, and in turn have gained the advantage of not needing to develop a specialized server to test the prototype kernel or run standard benchmarks.

## 4 Mach Internal Virtual Memory Architecture

In order to understand the required architectural changes, we consider the internal layers of the Mach VM architecture (see Figure 1).

### 4.1 Machine independent layer

Tasks contain threads, a virtual address space and an ipc space for port rights. The virtual address space is described by the ordered collection of map entries (`vm_map_entry`) in a virtual memory map (`vm_map`). Each map entry defines the mapping of a portion of a memory object (`vm_object`) to a range of the address space, including protection and inheritance attributes. Each memory object is a collection of untyped data organized into pages (`vm_page`). Pages within an object are identified by their offset from the start of the object, hence a map entry can be considered as a mapping from a portion of the task's address space to a portion of the object's offset space. The kernel manages physical memory as a cache of memory object data; the actual data

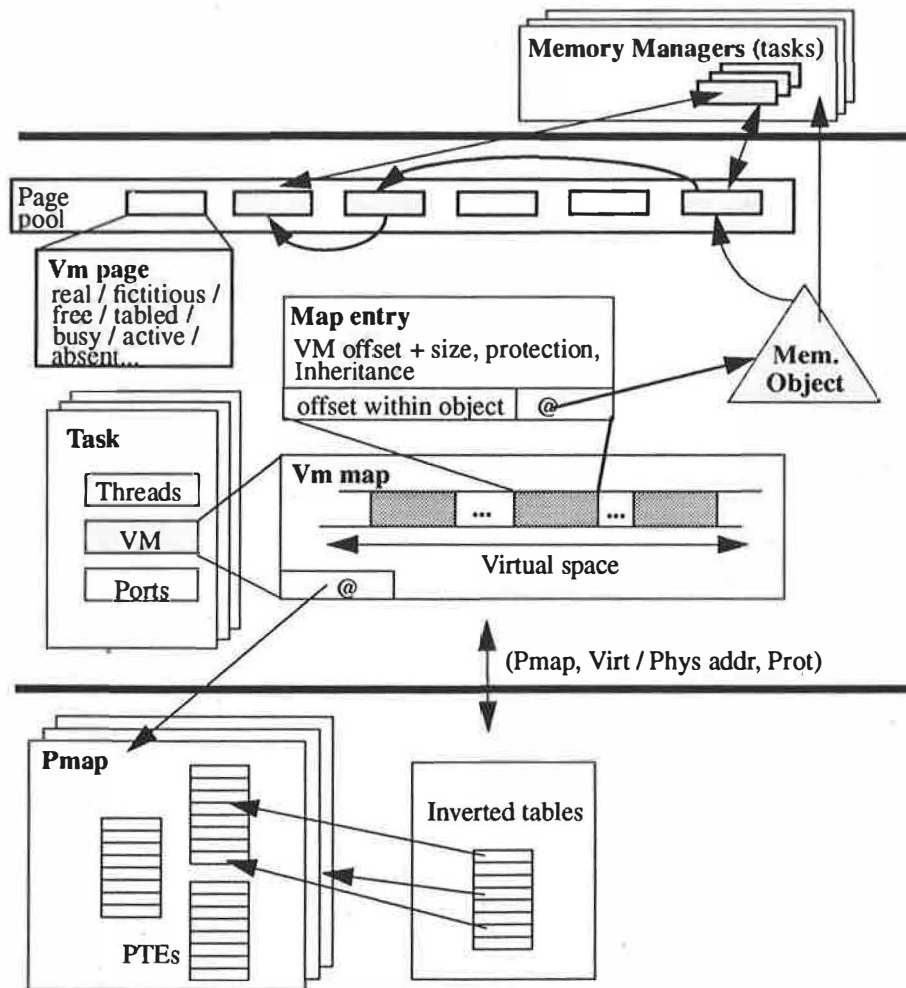


figure 1: Mach VM Architecture

is managed by an external memory manager or the kernel's default memory manager.

Memory managers are tasks like any other. The kernel requests memory object pages from the related memory manager at page fault time, and sends the data back to the memory manager on request from the manager or at pageout time. Memory object pages correspond to physical pages when the data is in memory. Fictitious pages (no corresponding physical page) are used when the data is not present, but has been requested from the pager. Anonymous memory, created as the result of a `vm_allocate()` call or copy operation, is managed by the default memory manager. This manager also acts as the pager of last resort when external managers do not free pages as requested by the kernel. The default memory manager's memory is wired down as it cannot page itself.

Memory objects can be shared by multiple tasks or mapped more than once within the same task. When a task sends data to another task, or when a task inherits memory from another task without sharing it, the memory is (logically) copied into a new object. To improve performance, the VM system avoids copying the object's data by using shadow objects. A shadow object contains only the pages modified since the copy, with the original object providing the unmodified pages. Pages in the shadow object are said to shadow (i.e., cast a shadow on) the corresponding pages in the original object that they replace; a light shown from above would illuminate only those pages visible to the task.

## 4.2 Machine dependent VM layer

All the information required to find the memory page corresponding to a virtual address is in the machine independent layer; no machine-dependent data structures are used. Once this information is located, it is necessary to instruct the hardware that a virtual address in this task must be mapped to a specific physical page with appropriate protections. This is achieved via a simple interface to the physical map (pmap) layer. The pmap layer is responsible for managing the memory management unit, including installation, modification, and removal of virtual to physical translations. The pmap layer must also be able to tell the machine independent layer if a given physical page is mapped or if it has been modified. As a consequence, pmap modules for hardware that employs a tree-structured or linear page table must also manage an inverted list to enable it to locate all virtual mappings of a physical page.

## 4.3 IPC

Mach's IPC system is relevant to our work because it uses the VM system when large amounts of data are sent in a message. Mach IPC messages are typed, providing information to the kernel that allows it to determine if the message contains either of the following:

- Port rights: Messages are the only way to transfer these kernel managed entities among tasks.
- Out of line memory (passed by reference).

When sending data via messages, a task can either send it *in-line* or *out-of-line* (ool). All data sent in a message has copy semantics; the receiver appears to get its own copy. For in-line data, this is implemented by copying the data into the message body at send time and copying it from the message body at receive time. Out-of-line data is passed by reference, and the kernel employs lazy-evaluation and copy-on-write techniques to minimize the amount of data that must be copied to implement copy semantics.

The VM layer provides an abstraction called a map copy that the IPC layer can manipulate. A map copy is a temporary repository for data being sent via a message. It is created when the message is sent and destroyed when the message is received. The map copy abstraction is a logical copy of the virtual region passed in the message at the time the message was sent.

There are three different possible representations of a memory region in a map copy:

- A vm object
- A list of pages
- A list of vm\_map\_entries

The object representation is used for pageout. When the kernel sends a page to a pager, the page is moved to a new object to allow it to be backed by the default pager, thus protecting the kernel from a recalcitrant pager that refuses to perform pageouts. The new object contains a single page, and it is more efficient than using map entries for this case.

The list of pages is used for pagein and device operations. It allows the kernel to move pages directly from the original object if the sender deallocates the data at send time. For most pageins and device reads, the sent data is deallocated once sent by the pager or device handler.

The map\_entry type is the most common. When a thread sends a region of virtual memory in a message, the kernel copies the map entries and mark the object(s) as copy on write. To facilitate the implementation, the VM layer splits and clips together vm\_map entries to match the region. At receive time these map entries are moved into the receiver's map, providing the receiver with copies (shadows) of the original objects (see Figure 2).

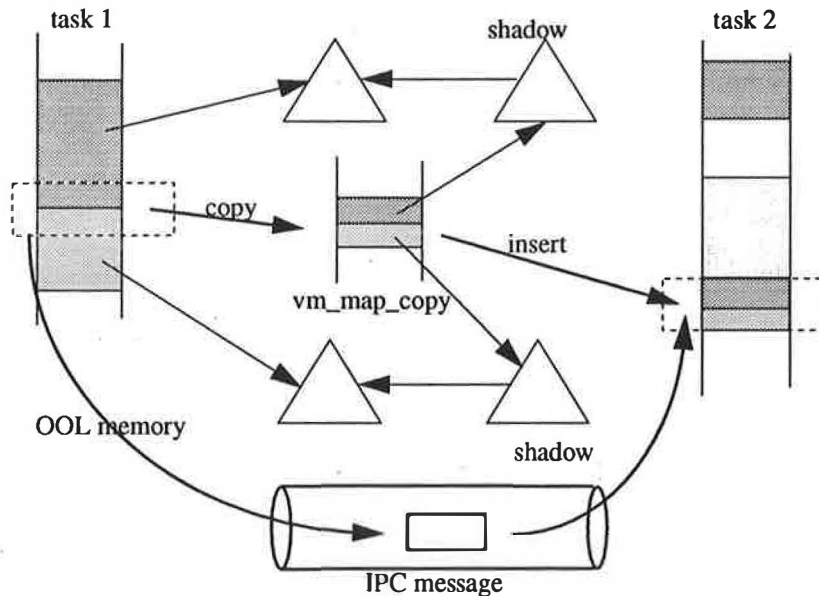


figure 2: Sending OOL Memory Through Mach IPC

## 5 Architectural changes

This section describes the architectural changes made to Mach's virtual memory management system to support real memory hardware.

### 5.1 Object / Segment / Page relationship

An object contains a linked queue of referenced (present or being requested) pages. For the real memory system, the pages of an object must be contiguous and in existence (it is not possible to use lazy evaluation to fault them in). Thus an object represents a single physical segment. A virtual segment in a task's address space is a different concept; it may be mapped to an entire physical segment, or only a portion thereof. In turn, physical segments may be mapped multiple times into the address spaces of multiple tasks.

The `vm_page` entity persists in the real memory design, as it is the unit of communication with external pagers and the IPC system. Every page of an object exists (unless the object is swapped), cannot be fictitious (i.e., it always corresponds to a physical page) and must be initialized before any access. The pages do not need to be linked into a queue as they are contiguous. The page abstraction is not used by and is not visible to machine dependent code.

Mach's original VM system makes very little use of object sizes. Sizes of externally managed objects are adjusted in response to page faults that appear to extend the object. This is unreasonable for a real memory system. Even if hardware permits restarting an access beyond the end of a segment, this would require expanding that segment for each new page. If the next sequential page is not free in physical memory, reorganization and compaction of physical memory are required to accommodate the newly enlarged segment. The real memory system initializes the object size at object creation time to avoid these inefficiencies.

Contiguous physical pages (i.e., a physical segment) must be allocated and initialized at memory allocation or mapping time. Because access faults may not be restartable, the real memory system cannot rely on the kernel to detect initial references to pages. Hence, memory allocation causes the resulting memory to be zero

filled immediately. Similarly, memory mapping causes the mapped pages to be requested from the object's memory manager immediately. The algorithm and code responsible for requesting pages from the memory manager or zero filling pages are basically unchanged.

Lazy evaluation also cannot be used to optimize memory copying (e.g., copy on write) because the real memory system cannot assume that write references can be detected and responded to. As a consequence, shadow objects cannot be used. Copying an object involves creation of a new regular object and a new segment to hold the copy; then relevant memory is physically copied. When a task is created, the objects corresponding to memory inherited as copy are copied by the kernel instead of being shared copy-on-write.

## 5.2 Map entry / segment relationship

Virtual and physical segments do not correspond one to one. A physical segment can be mapped multiple times at different addresses within the same or different tasks. Consider a program with text and data. The memory object (a file) corresponds to a single physical segment in memory. But the text and data have different protections, thus requiring two virtual segments (Figure 3a). Two virtual segments are also required if text and data are contiguous in the file but separated in the task's address space (Figure 3b). On machines with a single segment per task, the text and data virtual addresses must correspond to the text and data offsets within the memory object, and there can be no distinction in protection at the hardware level (Figure 3c).

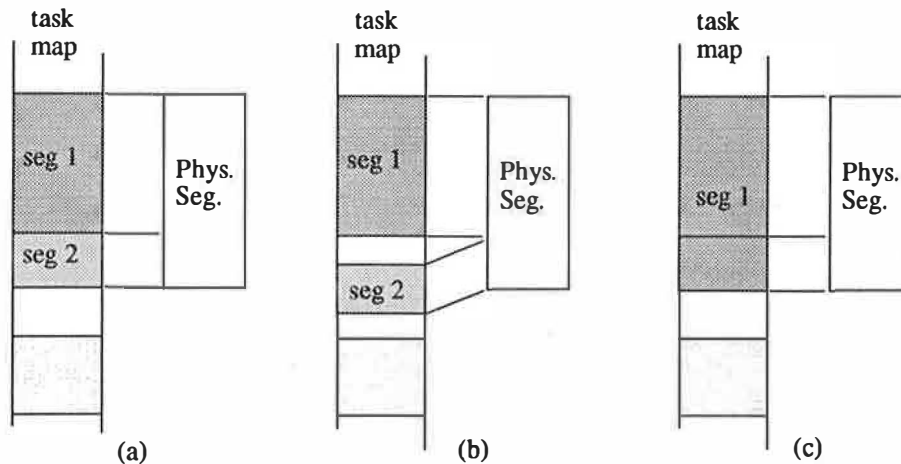


figure 3: Virtual and Physical Segments

In all three cases there are two map entries, even if they correspond to a single object and physical segment. The machine independent layer marks each entry with the appropriate protections, even if these protections are implemented identically by the hardware.

## 5.3 Anonymous memory management

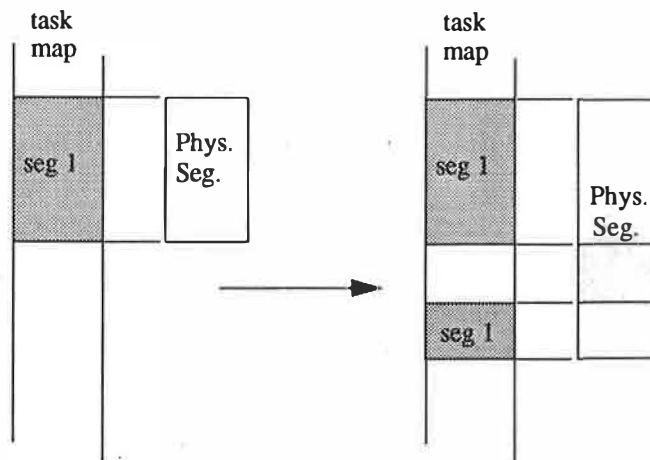
The real memory system attempts to avoid creating new segments (both virtual and physical) when anonymous (zero fill) memory is allocated. This is in contrast to mapping an externally managed object which requires creation of the segments. Mach's original VM architecture creates a new object in the following cases:

- The virtual memory region is not contiguous to an existing one (user specified virtual address)
- Required protection/inheritance differ from the one of the existing object
- Object reference count is greater than one.

These conditions can be made less restrictive in the real memory system by employing lazy deallocation and overly eager allocation. This results in accessible valid memory at locations that would be invalid in the original system, but this is a small price to pay for the resulting saving of virtual segments.

If the virtual address at which a user wishes to allocate new memory is not contiguous with the last virtual address of an existing anonymous memory region, the same memory object can still be used in some cases (Figure 4). The middle pages will be allocated in the physical segment but not used. Accesses to these pages will not be trapped even though the memory region is invalid from the memory system's point of view. The corresponding segment area could be uninitialized, but for security reasons, they should be zero filled (to be implemented in our prototype). The memory management system must prevent mapping of new objects inside this invalid memory region.

In the case where memory is allocated without specifying a fixed address for the new memory, the real memory system tries to merge the new region with an existing segment. The predefined read, write and execute protections of the new and existing region may not match. As an example the OSF/1 server sets the data region protection to r/w, but the default r/w/x protection is used when the kernel allocates memory within a task (e.g., for message passing). This prevents coalescing the newly allocated memory region with the data region. On most hardware, these 2 protections are identical. Hence, the real memory system introduces a machine dependent function to compare protections based on whether they are implemented identically (e.g., our prototype assumes that r/w/x is equivalent to r/w). In this case, a single physical and virtual segment suffice to implement the resulting protection.



**figure 4: Anonymous Memory Allocation**

Different concerns apply if the existing region to be extended has an object with a reference count larger than one. This indicates that the object is mapped into multiple tasks or may also be mapped by multiple regions of the same task. Regions are often split when a task deallocates memory in the middle of an existing memory region. In this case it is possible to coalesce newly allocated memory with this object even if the object's reference count is larger than one. Shadow objects no longer cause this situation, as they aren't used in the real memory system. In order to determine if an object is actually shared, the physical (object) segment data structure contains two fields indicating the task in which the segment is mapped and where. Reserved values are used to indicate that the object is mapped by more than one task or at more than one virtual address in the same task. These fields are also used by the segment management layer.

At deallocation time, if a task specifies a memory region in the middle of an existing object, the VM layer registers it and modifies the map entries, but does not change the task's segments. The deallocated region

remains accessible to the application and may be reused to satisfy a subsequent memory allocation request.

## 5.4 IPC

The real memory implementation changes the use of map copy structures by the IPC system. The map entry lists of a map copy always refer to regular objects. Object copies (of the part sent in the message) are performed at message send time. On the other hand, at receive time, the kernel allocates anonymous memory in the receiver's memory space and copies the data to minimize segment usage; directly mapping the copied object(s) into the receiver's address space would require one or more additional virtual segments. The kernel often splits map entries to facilitate the creation of these map copies. This appeared to pose a problem in minimizing the number of virtual segments, so code was added to the prototype to be aggressive about merging map entries. This code is not necessary because one virtual segment can encompass multiple vm map entries, but yields performance benefits by reducing the number of map entries. Our prototype does not support the page list version of map copies, in order to avoid problems if the pages in the page list are discontinuous. This is not strictly necessary, as the pages in a page list usually come from a single segment (and hence are contiguous), and could be copied if this was not the case.

## 5.5 Segment management

The segment management layer is new; it replaces and changes the functionality of the resident page management layer. The segment management layer implements segment allocation, deallocation and growth. Segment allocation requires fitting the desired segment into an available space in memory (in contrast to page allocation, which simply removes a page from a queue). Care must be taken to minimize fragmentation of the system's physical memory. Inefficient management of segment growth can drastically affect performance. Segment growth must be contiguous to the original segment, and so may require moving the original segment or its neighbors to make room. These segment moves are expensive and should be minimized. The pageout queue management code has been deleted because pageout is not a meaningful concept in a real memory system; entire segments must be swapped out instead (our prototype does not implement this). Objects can be still be marked as cacheable; this causes their physical segments to be cached for reuse, as in the original system.

Figure 5 shows the data structures used for segment management. Memory is divided into some number of contiguous segments; the number and their sizes vary dynamically as the system executes (for clarity we assume that the system's physical memory is contiguous, but this is not a requirement). Each segment is either busy (in use) or free (available for use). An allocation operation looks for a free segment of at least the requested size; this segment is split if it is larger than needed. All segments are organized into a doubly linked list by physical address to facilitate segment merging. Free segments are hashed by size and linked into buckets; adjoining free segments are always merged into a single segment. Our prototype uses a hash function that assigns segment sizes between adjacent powers of 2 to the same bucket (i.e., bucket  $n$  contains segments whose size satisfies  $2^n + 1 \leq \text{size} \leq 2^{n+1}$ ). The first segment of the appropriate bucket is selected by an allocation operation. If the segment must be split, the remaining portion is placed at the end of the free list for its bucket; this increases the likelihood that this portion will be available if the allocated segment should need to expand in the future.

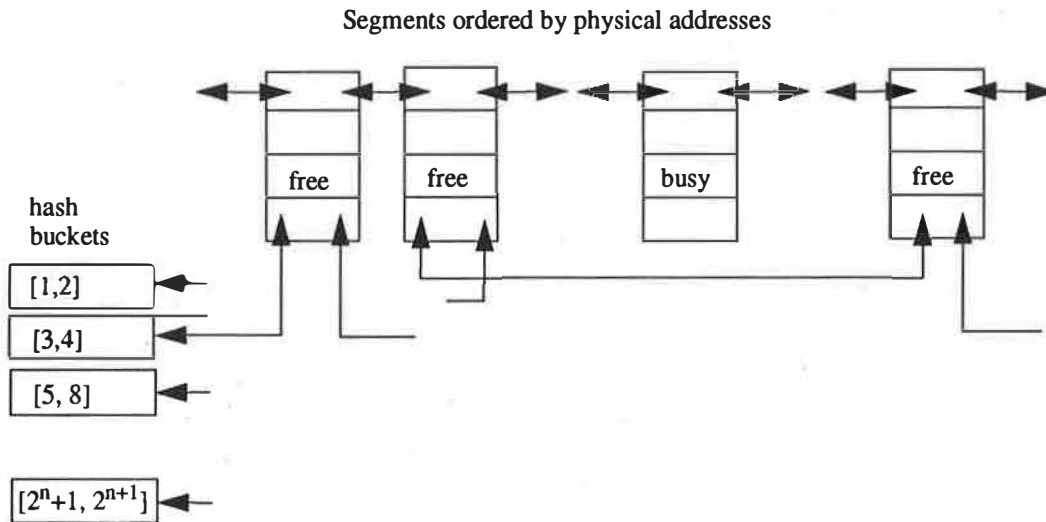
Each segment is represented by the following data structure:

```
struct seg {
    struct segment    *next_seg; *prev_seg;
    struct segment    *next_hash, *prev_hash;
    vm_offset_t       page_addr;
    vm_size_t         page_count;
    struct vm_page     *page;
    struct vm_object   *object;
```

```

    struct vm_map      *map;
    vm_offset_t        vaddr;
}

```



**figure 5: Segment Management Data Structures**

The `vm_page` field is used for communication with pagers and managing out-of-line memory sent via IPC; in order to save space in the segment structure, the `page_addr` argument could be obtained from the `vm_page` structure. The last three fields in the structure support segment growth. When moving a segment, the code checks that its `vm` object is not busy (e.g., `pagein`, `pageout` or device I/O in progress), and that the segment is not mapped multiple times. Once the segment is moved, it is remapped into the task that is using it. Additional locking constraints would be required for a multiprocessor implementation.

## 5.6 Management of the kernel task's space

We assume that the kernel runs in physical mode with the kernel's address space being the address space of physical memory. Multiple segments are not needed to protect the kernel from itself, and not all architectures of interest make multiple segments available for privileged mode execution. Running the kernel with physical addresses simplifies virtual space management because the physical segment addresses and the kernel virtual segment addresses match. This also makes it possible to move data directly from user-mode tasks to the kernel when desired because the physical address of the memory must be otherwise unused in the kernel's address space. Our prototype does not implement physical mode execution for the kernel; instead, we assume no limitations on the number of kernel virtual segments to simulate running in physical memory (segments are used for user-mode tasks).

A couple of minor modifications to the kernel's management of its address space were necessary. The kernel defines a special memory object, the kernel object, which is a repository for most of the data structures of the kernel that are not pageable. This object's size is initially zero, and is grown as needed. It is necessary to give this object a fixed size so that its segment can be managed like other physical segments. The prototype system does not implement automatic growth of the kernel object, although this is possible. The kernel may also define submaps that reserve portions of the kernel's address space. The primary use of these maps is to avoid deadlocks and related problems involved in paging and fault handling. Since our real memory system does not take page faults and swaps instead of pages, submaps were removed. In addition, reservation of address



space in this fashion would reduce the kernel's available address space and prevent it from accessing all possible segments by their physical addresses.

## 5.7 Pmaps

We did not address the issues involved in modifying this layer for real memory hardware because our prototype continues to run in virtual mode. Instead of invoking the pmap module with physical page addresses, the final system would provide segment addresses and sizes (e.g., to `pmap_enter()` and `pmap_remove()`). The segment number must be part of the virtual segment address. The distinction between the kernel and other tasks in current pmap modules must be maintained because the `pmap_enter()` and `pmap_remove()` calls do not need to change any mapping information for the (physically addressed) kernel in a complete real memory system.

The prototype contains no support for enforcing hardware restrictions on segment size and alignment. Additional machine-dependent data structures would be needed along with an interface to access them when searching for available address space in a `vm_map`. This internal interface would have little impact on the relevant algorithms. A similar modification was used by Wheeler and Bershad to optimize mapping alignments for support of virtual caches [9].

## 6 Server modifications

Use of virtual memory in the OSF/1 MK system is a potential obstacle to moving the server and commands to a real memory system. The OSF/1 server is a large user mode task whose memory is subdivided into many regions. At the time of this work, the server allocated over 30 megabytes (MB) of virtual address space to itself using between 50 and 100 map entries. A straightforward transformation of this to our segmented model would imply between 50 and 100 segments, which is clearly excessive. Between the emulator[2] and the use of shared libraries, OSF/1 commands required at least 2MB of virtual space and used between 12 and 20 map entries. The usage of segments and virtual space must be reduced to match the hardware constraints of a real memory system. Our primary goal was to enable execution on systems with hardware support for three segments; text, data, and stack (this corresponds to two segments on systems that simulate the stack in software because the stack is then part of the data segment). This section describes the minimal modifications to the system that were necessary to achieve both this goal and a sizeable reduction in virtual memory usage; we were pleasantly surprised by the ease with which this was accomplished.

The first step was to reduce the server's configuration by:

- Reducing the number of users from 16 to 2 (thus changing NPROC from 168 to 36)
- Removing the streams subsystem, and the AFS and system V filesystems.
- Limiting the memory allocated to mbufs to 256 K instead of 1 Meg.
- Reducing the maximum buffer cache size to 256 K.

The server's DEFAULT configuration is built with a shared memory area between the emulator and the server. Each task (emulator) shares three regions with the server: two for partially mapping the `u_area`, (one is read only) and a one to exchange data between the emulator and the server. This improves performance by reducing the number of RPCs and the amount of memory passed in messages. In a real memory system, this would require 3 additional segments per task. We removed this memory sharing option from the configuration.

There were a number of other configuration changes. We eliminated the configuration option that allows the server to map in a kernel maintained time value for faster access, saving the virtual segment that would be required to implement this. We changed the default stack size for user programs to 64KB from 2MB to reduce memory usage. Shared libraries consume virtual segments on a real memory system (at least one per library); we used statically bound commands and libraries instead to avoid this usage.

These changes allow us to achieve our segment reduction goals. The resulting server has three virtual segments (text, data, and stack). The text and data are the same physical segment, but different virtual segments for protection reasons. The stack segment is used only in initially starting the server; the cthreads package within the server allocates its own stacks from the data segment. Commands and other applications now have six segments: text, data, stack and 3 segments for the emulator (text/data/bss). A version of the server that does not use an emulator is currently being implemented[5]; with such a server, each user task would only require 3 segments. The current version of this server maps arbitrary regions of memory from its tasks into its address space to optimize the performance of moving data to and from them (copyin(), copyout()). This server also has a configuration option that uses kernel calls (vm\_read() and vm\_write()) to perform this function; a real memory version of this server would be configured in this way.

Our prototype has reduced the server's use of virtual address space to 4MB. The OSF/1 server can create up to 50 cthreads. In single user mode, there are already 30 cthreads. Each cthread is created with a stack size of 64K, accounting for almost half (about 2MB) of the virtual space usage. This stack size could be reduced to 32K saving another megabyte if desired. Further reductions are complicated by the use of buffers allocated from the stack to send and receive messages whose size may be up to 8KB. Recent changes to MiG have partly alleviated this restriction, but are not incorporated into our prototype.

## 7 Prototype Status

Our prototype kernel executes on an IBM-compatible PC with 16 Mbytes of memory, a 300 MB disk and an ethernet connection. The kernel code is based on NORMA\_MK11, but we have made no attempt to preserve NORMA functionality. In the memory management area, 11 files were modified and 2 new files were created. Modifications were also required to 13 other files in various areas of the kernel. Note that this prototype still executes in virtual mode, so there is very little change to i386-specific code. The vast majority of in-kernel interfaces are unchanged. Only the interfaces between the vm\_object layer and the resident memory layer have been changed to deal with segments. Submaps have been removed. The kernel memory allocation interfaces are unchanged, but the vm layer will always provide non-pageable memory.

The OSF/1 server described above comes up multiuser on the prototype kernel, and the resulting system is accessible via the network. The only programs that fail are the ones requiring more than 64 K of stack (e.g., rsh) or requiring large virtual space (e.g., gcc: 16 Mbytes). Since the prototype does not implement swapping, it stops when physical memory is exhausted.

## 8 Performance

It is important to note that the prototype is neither complete nor optimized. The performance impacts from completing and optimizing it may not be negligible. The pmap management code is still page based and still manages an inverted page table. The kernel task runs in virtual mode and invokes pmap subroutines. In addition, the cost of memory translation is still present. In various places we did not have the time to optimize the code, especially when paging in or reading pages from devices, where we could move segments from the sender to the kernel at copyin time.

All of the measurements reported here were done on a BULL BM600 PC:

- 25 Mhz i386
- 16 Mbytes memory
- 300 Mbytes ESDI drive
- WD8003 Ethernet board

A page copy (4096 bytes) takes approximately 1 millisecond.

The OSF/1 server was running in multiuser mode, with statically bound commands and libraries. The kernel and server configurations of interest are:

STD+WS:	Standard Mach micro kernel (NORMA_MK13)
STD+WS-vm	Real memory Mach microkernel(NORMA_MK11 based)
DEFAULT	Standard OSF/1 Server (SVR99)
SMALL	Reduced configuration server (Small buffer cache, no mapped u_area, no mapped time).

We address the following topics:

- RPCs
- VM
- disk and network I/O
- OSF/1 system call path
- Unix commands
- OSF/1 fork/exec

Attempts to measure program builds failed because the gcc compiler relies on a large virtual space; it unconditionally allocates 16 Mbytes of memory, which is more than the total available memory. The I/O performance figures must be analysed with care, as most of the elapsed time is “wait/io” time, and the extra page copies implied by the real memory kernel are not significant by comparison. However, in the case of a time sharing machines, the extra page copies would have a visible impact on performance. A user load benchmark like AIMIII would exhibit this effect.

## 8.1 RPCs

We used the standard *machipc test* program. This program is made up of one server and one client. They register and establish communication using the name server, in our case *snames*. Machipc uses the mach\_msg interface with the MACH\_SEND\_MSG and the MACH\_RCV\_MSG options. With the out of line (ool) option, the buffers are not read or written by either the server or the client.

Table 1: RPC performance (μsec/RPC)

Configuration	Null RPC	4 KB RPC	8 KB RPC	16 KB RPC
STD+WS / SMALL	310	730	730	730
STD+WS-vm / SMALL	290	2640	4320	7930
Relative performance (%)	1.07	0.28	0.17	0.09

Since the data is never accessed by the user programs, the out of line pages are never copied by the system with the standard kernel. The real memory system must copy the pages twice; the cost of these copies (1ms per page) accounts for most of the performance differences between the systems in these cases.

## 8.2 VM

We used 2 programs. The first one is a program that was written to measure the cost of zero-fill page faults (uninitialized page accessed for first time) and lazy evaluation faults (resident but not mapped page). The program allocates 1 megabyte of memory and accesses every word (4 bytes) of it twice. This gives us the cost of a zero-fill page fault and the cost of an already mapped access. Then the program forks and accesses every

word again, which gives us the cost of an unmapped access.

**Table 2: VM Performance: Memory access - ( $\mu$ sec/page)**

Configuration	zero fill	Lazy fault	Elapsed
STD+WS / SMALL	508	196	8984
STD+WS-vm / SMALL	0	0	9765
Difference	-508	-196	781

For the real memory kernel, although the program never takes page faults, the elapsed time is larger because the pages must be copied as part of the fork operation. The kernel with virtual memory support shares these pages copy-on-write instead.

The second program measures the cost of memory allocation and access time. It is run in two modes. The first mode allocates all the memory at once and then accesses each page. The second mode allocates one page at a time and accesses it.

**Table 3: VM Performance: malloc ( $\mu$ sec/page)**

Configuration	1 malloc + n touch	n malloc + n touch
STD+WS / SMALL	566	2109
STD+WS-vm / SMALL	390	1835
Relative Performance	1.45	1.15

The performance difference for the first mode is primarily due to the fault cost ( $\sim 190 \mu$ secs). The performance improvement for the real memory system in the second mode is somewhat surprising. The underlying segment must expand at every call, which may require segment copies and/or moves at various points during execution. The frequency and cost of these operations could be higher on a real memory system if physical memory is badly fragmented.

### 8.3 Disk I/O

These measurements utilize the OSF/1 file system. The program creates an 8 Mbyte file and then copies it to /dev/null.

**Table 4: Disk I/O Performance (Kbytes/sec)**

Configuration	Create	Copy to /dev/null
STD+WS / DEFAULT	450	394
STD+WS / SMALL	391	320
STD+WS-vm / SMALL	335	301
Relative Performance	0.74 / 0.85	0.76 / 0.94

The performance is compared to two server configurations. With the DEFAULT configuration, the emulator uses shared memory to pass data to the server. With the SMALL configuration, data is passed in messages. In the real memory case this implies 2 extra page copies

### 8.4 Ethernet I/O

We used ftp with an HP/700 workstation to send a large file to /dev/null on the HP/700 and receive a large file

from /dev/null on the HP/700.

**Table 5: Ethernet I/O Performance (Kbytes/sec)**

Configuration	Send	Receive
STD+WS / SMALL	110	200
STD+WS-vm / SMALL	100	180
Relative Performance	0.91	0.9

The file involved did not fit into the OSF/1 server's buffer cache.

## 8.5 System call path

We measured the getpid() and getuid() OSF/1 system calls.

**Table 6: System Call path performance ((syscalls/sec)**

Configuration	getpid()	getuid()
STD+WS / DEFAULT	13888	1538
STD+WS / SMALL	1562	1538
STD+WS-vm / SMALL	1666	1639
Relative Performance	0.12 / 1.07	1.06 / 1.06

There should be no difference in performance between the two systems with SMALL servers. Some of this difference may be due to variation involved in using the shell's time command instead of the gettimeofday and getrusage system calls. getpid() is considerably faster under the DEFAULT server because it takes advantage of the shared u\_area to avoid an RPC to the server.

## 8.6 Unix commands

We measured typical Unix shell commands:

```
$ find /usr/include/sys -type f -exec grep -i copyright {} \;  
$ find /usr/include/sys -type f -print  
$ grep -i copyright /usr/include/sys/*
```

The related directory contained 115 files. The command outputs were redirected to /dev/null to avoid measuring console I/O performance.

**Table 7: find/grep performance (elapsed seconds)**

Command	STD+WS / SMALL	STD+WS-vm / SMALL	Relative performance
find + grep	37.4	46.8	0.80
find	0.7	0.9	0.77
grep	24.3	24.5	0.99

The difference between the first and third lines is that grep is executed 115 times in the former. The find command time is negligible. This seems to indicate that the fork/exec mechanism is leading to this performance drop. The next measurement confirms this.

## 8.7 Fork/Exec/Wait

The test program loops on forking/execing and waiting for exec to return. The executed program is a null program: `main() {}`

Table 8: fork/exec/wait ( $\mu$ sec/call)

Configuration	$\mu$ sec/call
STD+WS / SMALL	49700
STD+WS-vm / SMALL	95100
Relative Performance	0.52

The program is 52 Kbytes in size and the null program is 22 Kbytes. In the real memory case, both the main and the null program are copied from the original cached object  $((52+22)/4 = 18.5$  ms). At fork time, both the text and the data are copied  $((52+64)/4 = 29$  ms). These times account for most of the performance difference.

## 9 Future work

The work described in this paper constitutes an initial prototype that demonstrates the feasibility of Mach as an operating system base for real memory hardware systems. There is still a fair amount of work to be done to create a version of Mach that runs on a real memory platform. The pmap interface needs to be redesigned to match the segmented memory model. This should result in a considerably simpler pmap module. The kernel needs to be adapted to execute in physical mode (versus the virtual mode of the current prototype). No change should be needed to our prototype's management of the kernel's (physical) address space to accomplish this. Segment swapping technology needs to be added; there is nothing new to be invented here, and a number of operating system implementations from which this technology could be adapted. In addition, there is a reasonable amount of code cleanup and error checking to be added, as would be expected with the first functional version of a prototype system. Additional extensions are needed to support multiprocessors and hardware-imposed segment alignment restrictions.

The prototype makes no attempt to optimize handling of out of line memory. Every virtual copy in the VM system is replaced by a physical copy in the real memory system. This results in a number of needless memory copies (e.g., a paged in page may be copied up to three times). In some cases, these are a matter of straightforward optimization work, but in other cases, kernel interface changes are needed. For example, a 'third-party' I/O capability in `device_read()` and `device_write()` that indicated the ultimate source or destination of the data in another task would avoid copies to and from the server's address space.

Some minor changes to the OSF/1 server are needed because certain virtual memory operations become quite expensive in a real memory system. The OSF/1 server knows that the kernel optimizes virtual memory operations. In traditional Unix implementations, the text portion of a task is often shared among the tasks running the same program. When a task needs to modify the text (e.g., for debugging), a private copy of the text is made. OSF/1 takes advantage of the fact that mapping an object as a copy of an underlying shared object is relatively cheap due to the use of copy on write techniques. But in the real memory case, mapping an object as a copy implies a physical copy of the object, with unfortunate (and unnecessary) performance impact.

## 10 Conclusion

The overall objective of this work has been to investigate the possibility of adapting the Mach micro-kernel to real memory hardware. We have shown that it is possible to do this without major modifications to the kernel interface. This result seems natural, since paged virtual memory functionality is a superset of

segmented memory functionality. In the recent past, researchers improved operating systems by using page-based virtual memory, and the present exercise has consisted of removing this improvement. Our performance measurements show that the absence of virtual memory is not always a handicap, and may be an advantage in some cases. This work has demonstrated the adaptability of the Mach kernel to yet another interesting class of hardware architectures.

## 11 References

- [1] D. Black, D. Golub, D. Julin, R. Rashid, R. Draves, R. Dean, A. Forin, J. Barrera, H. Tokuda, G. Malan, and D. Bohman, Microkernel Operating System Architecture and Mach, *Proceedings of the Usenix Workshop on Micro-kernels and Other Kernel Architectures*, Seattle, WA, April 1992, pp. 11-30.
- [2] D. Golub, R. Dean, A. Forin, and R. Rashid. Unix as an Application Program. *Proceedings of the 1990 Summer Usenix Technical Conference.*, Anaheim, CA, June 1990, pp 87-96.
- [3] INMOS Limited, The T9000 Transputer Products Overview Manual, First Edition 1991.
- [4] Open Software Foundation, OSF/1 Operating System Programmer's Reference, Prentice-Hall, Englewood Cliffs, NJ, 1991.
- [5] S. Patience, Redirecting System Calls in Mach 3.0: An Alternative to the Emulator, *Proceedings of the Third Usenix Mach Symposium*, Santa Fe, NM, April 1993, in this proceedings.
- [6] R. Rashid, Threads of a New System, *Unix Review*, Volume 4, August 1986.
- [7] R. Rashid, A. Tevanian, Jr., M. Young, D. Golub, R. Baron, D. Black, W. Bolosky, and J. Chew., Architecture-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures, *IEEE Transactions on Computers*, Volume 37, Number 8 (August, 1988), pp. 896-908.
- [8] R. Russell, The CRAY-1 Computer System, *Communications of the ACM*, Volume 21, Number 1 (January 1978), pp.63-72
- [9] B. Wheeler and B. Bershad, Consistency Management for Virtually Indexed Caches, *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, Boston, MA, October 1992, pp. 124-136.





# MIKE

## A distributed object-oriented programming platform on top of the Mach micro-kernel

Miguel Castro, Nuno Neves, Pedro Trancoso and Pedro Sousa

email: (miguel,nuno,pedro,pms)@sabrina.inesc.pt

INESC - R. Alves Redol n°9 1000 Lisboa PORTUGAL

### Abstract

*This paper describes the architecture and implementation of MIKE - a version of the IK distributed persistent object-oriented programming platform built on top of the Mach microkernel.*

*MIKE's primary goal is to offer a single object-oriented programming paradigm for writing distributed applications. In MIKE an application programmer can use C++ almost as he would in a non-distributed system.*

*The platform supports fine grained objects which can be invoked in a location transparent way and whose references can be exchanged freely as invocation parameters. These objects are potentially persistent. MIKE supports the abstraction of one-level store, persistent objects are transparently loaded on demand when first invoked and saved to disk when the application terminates. Class objects are special persistent objects which are dynamically linked when needed. The platform also offers distributed garbage collection of non-persistent objects.*

*This paper discusses how MIKE makes use of Mach's features to offer the functionality described above and the techniques used to achieve good performance. MIKE is compared with the UNIX versions of IK to evaluate the benefits of using Mach abstractions.*

## 1 Introduction

Writing distributed applications is still a difficult task due to the lack of adequate support. In a traditional environment a programmer uses a general purpose programming language to program most of the objects and an Interface Definition Language (IDL) to generate stubs to access remote ones. Generally, the IDL concepts do not match those of the programming language – remote objects are named and accessed in a different way, parameter passing is different and inheritance does not exist or has different rules. All these problems tend to separate the application into static sets of co-located objects that communicate with remote ones using Remote Procedure Call (RPC) based client-server interfaces.

MIKE, Mach IK Environment, is a redesigned version of the INESC Kernel (IK [9, 13]) that takes advantage of Mach's [1] features. IK's primary goal is to offer a single object-oriented programming paradigm for writing distributed applications. In the current versions, applications are written in EC++, a language with the same syntax as C++ but

with some restrictions and semantic extensions [11]. It supports fine grained objects which can be invoked in a location transparent way across the distributed system and whose references can be freely exchanged as invocation parameters. An application can be designed as a set of fine grained communicating objects whose location in the system can be dynamically set at run-time.

Traditionally, a programmer uses files to persistently store objects and has to handle the conversion of all context dependent data (e.g. pointers) when saving or retrieving objects. IK solves these problems by offering the abstraction of a one-level store – all objects are potentially persistent and persistent objects are transparently loaded when invoked and saved to disk when the application terminates. Class objects are special persistent objects which are dynamically linked to running applications when needed. A new version of a class object can replace an old one at run-time, provided it preserves the old interface and semantics.

IK's implementation uses only UNIX abstractions to achieve portability. It runs on a network of Sun workstations, PC and Bull machines running different UNIX flavors. MIKE shares its goals and model with IK but bases its implementation on Mach abstractions. The MIKE prototype is running on the Mach 3.0 micro-kernel and the BSD single server.

There are several demonstration applications running, from which we outline a distributed ray-tracer. This ray-tracer is designed as a set of MIKE objects which implement the processors farm model. These objects are distributed through several nodes to achieve true parallelism and speed-up image rendering.

## 2 Overview

MIKE uses an object-oriented approach, all entities are objects conforming to a simple conceptual model. This model was inherited from IK and it is fully described in [13].

### 2.1 Basic abstractions

The platform offers the following set of basic abstractions:

**Object** – An object is a passive data structure which exports a set of methods defined by its `class` (or by its class superclasses). Objects can be volatile or persistent. Persistent objects survive application termination. Objects can be invoked in a location transparent way.

**Activity** – An activity is an active object which represents a thread of control. Activity synchronization is based in lock and condition objects. Like all objects, activities, locks and conditions can be invoked in a location transparent way.

**Context** – A context is a protected address space mapping a set of objects. The context itself is represented by an object. This object has methods to create activities in its own address space. Several activities run in each context.

**Node** – A node is an object which represents a machine in the distributed system. Activities invoke a node object to create a context on its corresponding machine. Several contexts can coexist in the same node.

Mach offers abstractions that match these closely. A Mach port can be associated with distributed objects and send rights can be used as object references. Mach Inter-Process Communication (IPC) can be used to access remote objects in a location transparent way and the Mach interface Generator (MiG) stubs can be used to marshall and unmarshall invocation parameters. The context abstraction matches closely a Mach task – a resource container shared by several threads of execution. An activity can be mapped on a Mach *Cthread* and locks and conditions can be mapped on *mutexes* and *condition variables* from the Cthreads package. These observations guided our implementation on top of Mach.

## 2.2 Architecture

MIKE runs on several computing nodes connected by a network and running the Mach 3.0 NORMA kernel<sup>1</sup> and the BSD single server. The platform is composed of a set of node servers, storage servers (SS), name servers (NS) and contexts. This architecture is depicted in figure 1.

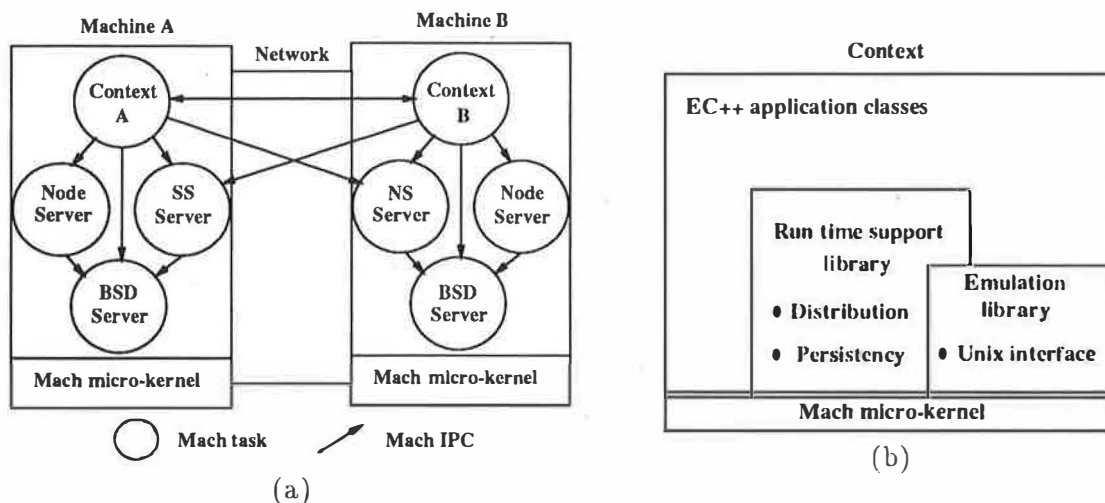


Figure 1: MIKE's architecture – possible configuration (a) and interface layering (b).

There is a node server on each machine. It boots the other MIKE servers on that machine, creates local contexts on behalf of activities on remote nodes and authenticates users.

The storage servers manage persistent objects' disk images. These servers are responsible for the generation of persistent object identifiers. They also offer a service that returns an object location given its persistent identifier.

The name servers cooperate to offer a distributed name service. This service associates names (strings) with objects. These associations are persistent.

The node, storage and name servers are multi-threaded UNIX processes. They use the UNIX functionality provided by the BSD server to access the file-system and communicate with the client contexts using Mach IPC.

EC++ application classes can use the UNIX interface or directly the micro-kernel interface. MIKE features are supported by the run time support library. Each context's process

<sup>1</sup>MIKE also runs in the non-NORMA kernels, but without distribution because the netmessage server does not work properly.

image is linked with this library. The run time support library offers transparent object invocation, dynamic linking, garbage collection of distributed volatile objects and cooperates with the servers to support object persistency. This functionality is implemented using mostly the micro-kernel interface.

The transparent object invocation abstraction hides the object location from the programmer. If an activity invokes a persistent object, which is in disk, the object is transparently mapped on the activity's context. Similarly, when an activity invokes an object mapped in a remote context invocation is transparently forwarded to that context.

A port is associated with each distributed object and remote invocation is based on MiG generated stubs. These stubs are encapsulated in EC++ classes which implement the mechanism described above. The MiG client stub is encapsulated in the client proxy class. This class exports the same interface as the original class. The server proxy class encapsulates the server stub. Each context has a port set where all object ports are inserted. A set of dispatcher threads wait for requests in this port set. They service remote invocation requests directed to the objects mapped in the context. The remote invocation mechanism is used to share objects between contexts.

### 3 Implementation

MIKE reuses part of IK's implementation, but distribution support is significantly different. MIKE also adds real multi-threading support and distributed garbage collection.

#### 3.1 Object structure, local references and object invocation

Instance objects and their classes are represented in memory by the structure shown in figure 2. Objects are represented by Run Time Headers (RTH). The RTH hold all relevant per-object information. Local object references are pointers to these headers.

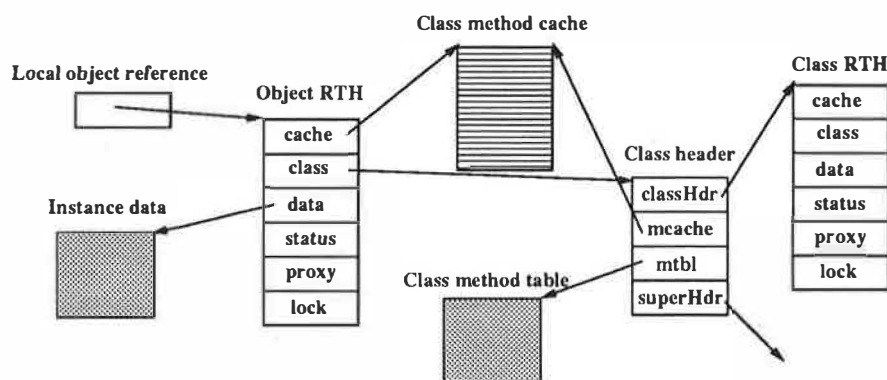


Figure 2: In memory object structure.

The **data** field in the RTH points to the object's instance data. This indirection simplifies object mobility. IK's conservative copying garbage collector [5] relies on this indirection. It also simplifies the implementation of lazy evaluation techniques like on demand dynamic linking and delayed pointer swizzling. On the other hand, accesses to the instance data are penalized.

The **status** field is used to indicate the object's type (e.g. whether the object is a class, a client proxy, a server proxy or a regular object) and its status (e.g. whether the object is mapped or unmapped).

The **proxy** field is used to store information relevant to the remote invocation process (see section 3.3).

The **lock** field stores the object lock, which is used to ensure mutual exclusion in RTH accesses. We chose to use a spinlock instead of a relinquishing mutex because the size of the first one is one fifth of the size of the second. Furthermore, our measurements show that a spinlock is more efficient than a mutex.

Method selection is performed by searching for the method in the class hierarchy, using a late-binding mechanism similar to the one used in Smalltalk and Objective-C. Each class has a method's cache which stores the entry points for the most recently invoked methods, speeding up invocation. The **cache** field of an object's RTH points to the method's cache of its class. The per-class method's cache is shared by several threads and accesses must be synchronized using the lock in the class RTH.

The detection of object faults (i.e. the invocation of unmapped objects) is achieved by trapping object invocations. This prevents direct accesses to the instance data. When an object is invoked, the invocation primitive looks for the method in the cache. Only if there is a cache miss is the object status tested to confirm whether it is a normal cache miss or an object fault. Hence, the RTH of unmapped objects always point to an empty method's cache.

The object fault handler tests the status field. If the object is persistent and not mapped anywhere, it retrieves the object from persistent store. Otherwise, if the object is a client or server proxy, the object fault handler creates an instance of the corresponding class. In either case, the object's class hierarchy is dynamically loaded and linked. Several object faults can be handled concurrently but not on the same object. Hence, the object fault is handled under the protection of the object lock.

## 3.2 Distributed object references

Most objects are known only inside a single context and are identified using only local references. When objects get referenced by persistent names or by other contexts they are assigned more expensive identifiers.

MIKE associates a Mach port with objects known remotely and remote contexts use send rights to reference them. This scheme has several advantages – location transparency, notifications of port destruction, port reference counting and protection by capabilities [3]; but it does not solve all problems.

### 3.2.1 Referencing potentially persistent distributed objects

On the one hand persistent objects survive the death of all contexts referencing them and, on the other hand, ports are volatile entities – a port is destroyed along with its associated rights when the task holding the receive right terminates. Therefore send rights are inadequate for use as distributed references to persistent objects. Persistent objects must be identified using persistent identifiers.

Since all objects are potentially persistent, using send rights as the only remote references to volatile objects is also problematic. If a volatile object gets promoted to persistent there is no easy way to transmit the new persistent identifier to all the contexts holding

send rights. This problem is solved by assigning a persistent identifier, *a priori*, to a volatile object, the first time a reference to that object is exported.

MIKE's persistent identifiers are called Low Level Identifiers (LLI). They are unique and persistent, i.e. the identifier is valid until the object it refers to is explicitly deleted. The LLI is sufficient to identify the object. However MIKE uses the tuple  $\langle oLLI, sRight \rangle$  as a distributed object reference to retain the benefits of using send rights. In this tuple  $oLLI$  is the object's LLI and  $sRight$  is a send right to the port associated with the object.

Remote invocations always try to use the send right. If it is invalid, the LLI is used to question the storage server responsible for the object and either get a new send right or map the object locally (see section 3.5.2).

### 3.2.2 Lazy port allocation

Associating ports with objects has all the advantages enumerated above but can consume a significant amount of kernel resources. Creating a port for every object is not feasible and it is also inefficient; because the number of objects is usually very large and most of them are short lived and never become known outside their creation context (e.g. one of our test applications, a cooperative document editor, creates 250 objects per second).

We use lazy evaluation techniques to minimize this problem – a port is only associated with a mapped object the first time a reference is exported. Furthermore, if the object is persistent and is currently not mapped, no port is associated with the object, only the LLI is sent. This significantly reduces the number of ports used.

### 3.2.3 “On-the-wire” reference representation

A context references a remote object using the tuple  $\langle oLLI, sRight \rangle$ . However, to invoke the object, the LLI of the object's client proxy class is also required. Hence, a reference is transmitted as the tuple  $\langle oLLI, sRight, cpLLI \rangle$ , where  $cpLLI$  is the client proxy class' LLI, whenever all the information is locally available. This default policy can be changed on a per-class basis to enhance performance.

### 3.2.4 Exporting object references

The primitive used to export references (**XRef**) converts local object references into the tuple  $\langle oLLI, sRight, cpLLI, pType \rangle$ . The first three components are the “on-the-wire” reference representation and  $pType$  is the port type.  $pType$  is needed because object ports are declared polymorphic on the sender's side [8]. The proxy classes call **XRef** for each exported reference.

**XRef** tests the object's status. If the object does not have an associated server proxy, **XRef** associates an LLI, a port and an instance of the corresponding server proxy class with the object. The port and the server proxy's RTH are allocated together to ensure that the port name matches the RTH's address. Collisions are rare enough for this process to be efficient and it saves a hash table and speeds up message dispatching.

The port name is stored in the **proxy** field in the object's RTH and the object's port is inserted in the context's port set. The **cache** field in the server proxy's RTH is initialized

with a pointer to an empty method's cache and the object reference is stored in the `proxy` field.

Since distributed garbage collection is based in no-more-senders notifications, the send right for a locally mapped object must be generated using `mach_port_insert_right` under the protection of the object's lock. This lock synchronizes reference exporting with the no-more-senders notification handler. The value of `pType` returned in this case is `MACH_MSG_TYPE_MOVE_SEND`. On the other hand, if the local reference points to a client proxy's RTH, `pType` is returned with the value `MACH_MSG_TYPE_COPY_SEND`. Therefore, the extra send right is generated by the `mach_msg` call avoiding the extra system call.

### 3.2.5 Importing object references

The primitive used to import references (`IRef`) converts the tuple `<oLLI, sRight, cpLLI>` into a local reference. The proxy classes call `IRef` for each imported reference.

`IRef` starts by searching for a RTH associated with `oLLI` in a hash table. If the search fails a RTH is created and associated with the LLI. This association is registered in the hash table. Once again, the RTH's cache field is initialized with a pointer to the empty method's cache.

When `sRight` is valid, `IRef` must handle the extra user reference to the send right. If the object is mapped locally, the send right received must be destroyed to enable no-more-senders notifications. On the other hand, if `sRight` corresponds to a locally mapped client proxy the reference can be discarded lazily – a counter is incremented and the extra references are discarded when it reaches a high-water mark.

If `sRight` is valid the object it refers to is mapped somewhere. Therefore, if the object was not known yet `cpLLI` and `sRight` are saved in the `data` and `proxy` fields of the RTH and the RTH is marked as belonging to a client proxy.

## 3.3 Remote object invocation

In a distributed object-oriented system like MIKE, remote object invocation is a fundamental primitive. The programmer must be able to program and invoke remote objects like local ones with acceptable efficiency. If the invoking object and the invoked one are co-located, invocation proceeds locally and without interposition of any proxy object, otherwise a remote invocation mechanism is used.

The remote invocation mechanism relies on three objects:

- An instance of the client proxy class mapped in the remote invoking context. This instance holds a send right to the port associated with the remote object it represents.
- An instance of the server proxy class mapped in the context of the invoked object which represents all the contexts holding references to the object. It holds a receive right to the object port and a reference to the object.
- The object itself mapped in the invoked context. Its class knows nothing about communication and remote invocations.

A remote invocation begins when the client proxy is invoked. The client proxy calls the MiG client stub. They marshall the invocation parameters and send an IPC to the

remote object port. In the remote context, the dispatcher thread finds the appropriate server proxy and invokes it. The server proxy and the MiG server stub unmarshall the invocation parameters and invoke the object. Control is returned to the client context through the same path.

### 3.3.1 Proxy class generation

The code for the proxy classes is automatically generated from the original class definition. This process is depicted in figure 3. All the actions are performed by a shell script, in a way transparent to the programmer.

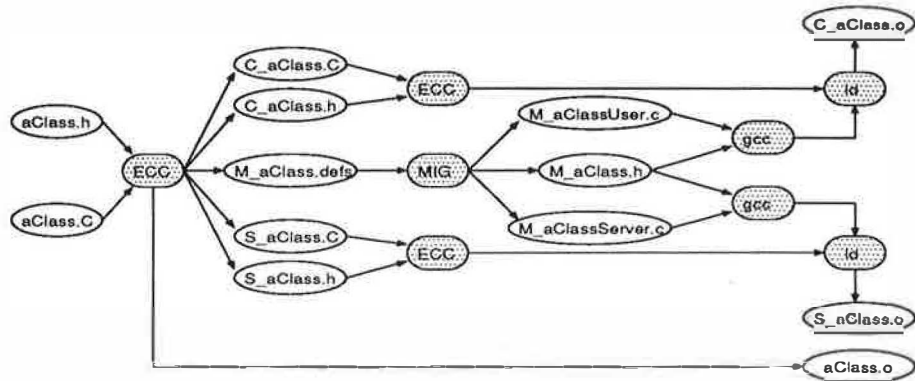


Figure 3: Proxy class generation. ECC is the EC++ compiler.

In the first step ECC generates the client proxy class (C\_aClass.[Ch]), the server proxy class (S\_aClass.[Ch]) and the MiG definition file (M\_aClass.defs). In the second step the proxy classes are compiled and the MiG definition file is processed. The third step compiles the MiG output files and links them with the corresponding proxy class. In the last step (not shown in the figure) class objects are created and an LLI is associated with each class.

### 3.3.2 The proxy classes

To describe the proxy classes we use class *aClass* as an example:

```

struct aStruct {
    char aChar;
    int anInt;
};

class aClass : public bClass {
public:
    virtual char aMethod(aStruct, int*, object*);
};
  
```

This class definition is processed as described previously.



The MiG definition file generated is as follows:

```
subsystem M_aClass 5;

userprefix call_;
serverprefix do_;

routine aClassaMethodF7aStructPiP6object
(
    port : mach_port_t;
    in    a1_1 : char;
    in    a1_2 : int;
    inout a2 : int;
    in    p3o : mach_port_t = MACH_MSG_TYPE_PORT_SEND;
    in    l3o : lli_t;
    in    l3c : lli_t;
    out   r : char
)
```

In order to support inheritance in remote invocations, the *message base id* of the subsystem is given by the number of non-pure virtual methods defined in the direct and indirect base classes of the subsystem's corresponding class. Note that multiple inheritance is not supported.

The subsystem defines a MiG routine per each method in the original class. The routine has the same parameters as the method but object references are expanded into their "on-the-wire" representation and structures are recursively decomposed into their constituent elements. The method's return value is also added as a routine parameter.

Parameters passed by value are mapped onto MiG in parameters. If they are passed by pointer they are mapped onto *inout* parameters, to cover their worst case use. Note that call-by-value semantics are respected. However, call-by-reference is only respected with object references; with basic types, structures and arrays it can only be emulated by call-by-value-return.

Each routine's name is obtained by concatenating the class name, the method name and the method signature. This is needed to ensure the name resolution rules of C++.

The client proxy class generated in this example is as follows:

```
class C_aClass : public C_bClass {
public:
    virtual char aMethod(aStruct, int*, object*);
};

char C_aClass :: aMethod(aStruct a1, int* a2, object* a3)
{
    char r;
    LLI_t l3o, l3c;                // object and class LLI
    mach_port_t p3o;               // object port
    mach_msg_type_name_t t3;       // port type
    XRef(a3, &p3o, &l3o, &l3c, &t3);
    mach_port_t p;
    getPort(this, p);
    handleError(call_aClassaMethodF7aStructPiP6object(p, a1.aChar, a1.anInt, a2,
                                                       p3o, t3, l3o, l3c, &r));
    return r;
}
```

The methods of the client proxy class decompose structures into their constituent types (and later compose them if the structures are passed by pointer or by reference). This step is important to guarantee the correct typing of the message which, according to the Mach philosophy, can be used to handle different data representations with a receiver-makes-it-right policy. This decomposition also eliminates dependencies on the compiler "alignment inside structures" rules. Finally, this analysis is important to detect embedded object references or pointers which must be handled separately.

The macro `getPort` retrieves the remote object's port name from the proxy field of the client proxy's RTH.

`handleError` is a macro which calls the method `handleChildDeath()` in the event of failure of the context where the remote object was mapped. This method is defined in the base class `C_object`. It tries to map the object locally or obtain a send right to the new object port. It can be redefined to provide a per-class or per-object child-death handler.

This example's server proxy class is as follows:

```
class S_aClass : public S_bClass {
public:
    virtual void dispatch(mach_msg_header_t*, mach_msg_header_t*);
};

void S_aClass :: dispatch(mach_msg_header_t* in, mach_msg_header_t* out)
{
    if (M_aClass_server(in, out))
        return;
    else S_bClass::dispatch(in, out);
}

extern "C" {
kern_return_t
do_aClassaMethodF7aStructPiP6object(mach_port_t p, char a1_1, int a1_2,
int* a2, mach_port_t p3o, LLI_t l3o, LLI_t l3c, char* r)
{
    struct aStruct a1;
    a1.aChar = a1_1;
    a1.anInt = a1_2;
    object* a3;
    IRef(l3o, p3o, l3c, a3);
    aClass* obj;
    getObj(obj, p);
    *r = obj->aMethod(a1, a2, a3);
    return KERN_SUCCESS;
}
}
```

This class exports only the method `dispatch`, but its code module also defines the functions called by the MiG demultiplexer function. These functions are responsible for composing the structures received; importing references; invoking the corresponding method on the right object; decomposing the structures passed by pointer or reference and exporting references.

The object on which to invoke the method is selected using the macro `getObj`. As we have noted, the address of the server proxy's RTH matches the port name. Therefore, the macro locates the server proxy object using the port name and retrieves the object's reference from the server proxy's RTH.

One advantage of interposing proxy classes in remote invocations is that the programmer can change the default implementation to meet specific application needs [12] (e.g.

introduce caching for efficiency, encapsulate shared memory access and manage replication). Furthermore, different client proxy classes can be used to access the same remote object.

The layering of the proxy classes on top of MiG stubs eases the interconnection of MIKE's applications with existing servers. The server's client stub can be encapsulated in a client proxy class and the server can be handled as an always remote MIKE object. For example, references to the server "object" can be freely exchanged in invocations. In spite of this advantage, MiG stubs are large. The use of a dynamic marshalling and unmarshalling mechanism similar to the one used in [6] would halve proxy class sizes.

### 3.3.3 Inheritance in remote invocation

Inheritance in remote invocation is supported by the addition of two shadow replicas of the original class hierarchy, one for the server proxy classes and another for the client proxy's. The class hierarchy for the example we have been presenting is shown in figure 4.

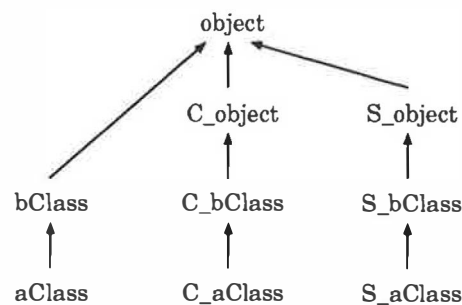


Figure 4: Class hierarchy which supports inheritance in remote invocation

The client proxy hierarchy ensures that they inherit the same interface as the original classes. The server proxy hierarchy guarantees that the base classes are all loaded when **dispatch** is invoked in a given class. As shown in this example, the method **dispatch** calls the MiG demultiplexer function. If the message identifier falls out of range, it calls the method **dispatch** of its base class.

This scheme only supports single inheritance and all methods must be virtual. Note that **C\_aClass** can be used wherever **aClass** is expected without the need to define the common interface in a base class. This is possible because MIKE supports a late-binding mechanism similar to Smalltalk's, it would not work in plain C++.

### 3.3.4 The Dispatcher

Each context has an instance of class **Dispatcher**. Its function is to receive remote invocation requests and call the correct method on the correct object to service them. The **Dispatcher** manages a set of service threads that concurrently wait for remote invocation requests on a port set. All the object ports are inserted in this port set.

The number of service threads is adjusted dynamically to adapt to varying load. The **dispatcher** object has two counters – **totalThreads**, which counts the total number of service threads, and **freeThreads**, which counts the number of threads waiting for

requests. These counters are updated concurrently by the threads under the protection of a lock.

Initially only one service thread is created. This thread executes the method `Dispatch`, a modified version of `mach_msg_server`. When a message is received `freeThreads` is decremented. If it becomes null and `totalThreads` is lower than a maximum value, a new thread is created to execute method `Dispatch` on the `dispatcher` object and `totalThreads` is incremented. Then the request is processed.

After processing the request `freeThreads` is incremented. If it reaches a given maximum and `totalThreads` is higher than a minimum value, the thread sends the reply and destroys itself. Otherwise it sends the reply and blocks waiting for a request using a combined send and receive operation for increased efficiency.

This simple scheme has the hysteresis needed to avoid oscillations and will avoid thread creation overhead except in abnormal load peaks.

The request dispatching is very simple. It is achieved with the following line of code:

```
((S_object *) Req->Head.msgh_local_port)->dispatch(&Req->Head, &Rep->Head);
```

Where `Req` is the buffer with the request message and `Rep` the buffer where the reply message will be placed. The dispatching takes advantage of C++'s polymorphism and of port name and server proxy reference matching. Since the port set only contains object ports it is guaranteed that the obtained reference always points to a valid object.

### 3.4 Garbage collection

MIKE uses a slightly modified version of IK's garbage collector [5]. Small objects are recycled using an algorithm similar to generation scavenging. Garbage collection of large objects and run time headers is based on an incremental mark-and-sweep algorithm. Only volatile objects are recycled.

The garbage collector roots are the register states and stacks of all the threads in the context, the references in the instance data of persistent objects with an LLI and a set of object references called `refSet`. Register states can easily be obtained using `thread_get_state`, but finding the used portion of each thread's stack can be more difficult. When a thread is preempted while executing emulation library code, the stack pointer value saved in the register state points to an emulation library stack. The real stack pointer value is saved at the bottom of this stack. This situation must be detected and handled correctly.

Each context has a collector thread. This thread executes the collection algorithm independently from other contexts. Synchronization between mutator and collector threads is based on a stop-the-world policy. The collector thread suspends all other context threads before executing the algorithm. Mutual exclusion is used to guarantee that threads are suspended in a safe state. The collector thread acquires the locks which protect sensitive data structures before suspending the other threads.

Distributed garbage collection combines Mach's built-in port reference counting mechanism with the local garbage collection. When a server proxy is associated with an object (in `XRef`) both references are inserted in the `refSet`. Then a no-more-senders notification for the object port is requested to the kernel. The context who imports the reference gets a send right to the object port and an associated client proxy object.

Eventually, the client proxy object stops being referenced and is recycled by the local collector. When this happens the send right is destroyed. If it is the last client proxy

the context where the object is mapped will receive a no-more-senders notification. The notification handler compares the current receive right *make send count* with the one in the notification message [8]. This comparison is made under the protection of the object lock to ensure that no references are exported during the process. If the *make send counts* differ a new notification is requested. Otherwise, the server proxy and object references are removed from `refSet` and they become subject to the normal garbage collection process. The receive right is destroyed when the server proxy is garbage collected.

MIKE's distributed garbage collection, like any other simple distributed reference counting algorithm, does not collect distributed cycles.

### 3.5 Object persistency

From the programmer's point of view there is a one-level store, all objects are manipulated in the same way whether they are persistent or volatile, mapped or on disk. Persistency is a dynamic attribute.

Objects are created volatile. They are promoted to persistent when they become reachable from a reference on persistent store. An activity can explicitly save a reference to an object on persistent store using the name service to assign a name to the object.

A persistent object's image can be saved explicitly invoking the method `flush` on the object or implicitly when the context where it was mapped terminates. When the image of a persistent object is saved all the objects reachable from its references are also saved.

A persistent object's reference can be obtained given its name, using the name service, and then the object can be invoked and its references followed transparently. A persistent object is mapped in the first context that invokes it.

If a persistent object ceases to be referenced from persistent store it is turned volatile and can eventually be garbage collected.

Several system components cooperate to offer this view to the programmer. The name servers offer the persistent name service and the storage servers and the run time support library cooperate to save and retrieve persistent object's images.

#### 3.5.1 Persistent object naming and clustering

MIKE's persistent references are called LLI. The LLI is a tuple with three components ( $\langle SSid, BGN, GN \rangle$ ). The *SSid* (Storage System identifier – 16 bits) identifies the storage system container where the object's persistent image is stored and the storage server responsible for the object. The *BGN* (Base Generation Number – 32 bits) and the *GN* (Generation Number – 16 bits) identify the object within a container.

Since there is a potentially very large number of persistent objects we do not assign an LLI to all of them. An LLI is assigned to an object (if it does not already have one) only in the following situations:

- The name service is used to assign a persistent name to the object.
- A reference to an object is exported to another context. If this object was volatile and later ceases to be distributed (which is detected through the distributed garbage collection) the assignment is invalidated.
- If, when saving persistent objects, it is found that an object is reachable from more than one object with an LLI.

In our platform persistent objects tend to be fine grained, around 48 bytes average size, so we use an object clustering technique to obtain larger grained entities called clusters. These entities are then manipulated by the storage system.

A cluster is a subgraph of the global persistent objects graph. In this subgraph there is only one object with an LLI – the head of the cluster. The head of the cluster is the only object directly referenced from the exterior of the cluster and so the LLI of the head identifies the cluster.

The tail of the cluster is composed of all the objects in the private subgraph of the head, that is, those only accessible from the exterior of the cluster through references in the head's instance data. The objects in the tail are stored in depth first order.

In the current implementation the clusters are rebuilt by the run time support library every time the head is flushed or the context where they were mapped terminates. During this process object references are translated into cluster offsets, if they point to an object in the same cluster, or into LLI, otherwise. Then the clusters are delivered to the storage system which saves them on disk.

When a method is invoked on the head of an unmapped cluster the run time library asks the storage service to map the cluster given the head's LLI. If the request is successful the cluster is mapped and references are converted from their disk representation to pointers in a lazy way – only when each object is invoked.

### 3.5.2 The storage system

Clusters are stored in a distributed persistent object space partitioned into a set of storage system containers (UNIX directories). Each container is managed by a different storage server, which runs at the node where the file system resides. Each user has a container assigned, all persistent object clusters created by this user's applications are stored in that container. Each cluster is stored in a separate file.

The storage servers are multi-threaded. There is a set of service threads waiting for requests in a service port. The service port is registered in the netname server <sup>2</sup> as "SSssid", where *ssid* is the container's storage system identifier.

Each storage server maintains an object location table. This hash table is used to obtain binding information for mapped persistent objects. Each entry in this table stores the object's LLI, a send right to its port, its client proxy class' LLI and a send right to the object port of the context where the object is mapped. The storage server uses Mach's port destroyed notifications to clear entries in this table when the contexts where the objects were mapped terminate.

A context interacts with the servers through a proxy layer, which encapsulates MiG stubs. This layer is part of the run time support library. When an operation is requested on an object, the proxy layer extracts the field *SSid* from the object's LLI and uses it to obtain the server's service port from the netname server. After acquiring the service port the operation is requested to the server using RPC. The service port is cached for efficiency.

When a context requests the first LLI, the proxy layer obtains a *BGN* value from its storage server. The context can use this value to create  $2^{16}$  LLI without communicating

---

<sup>2</sup>In the non-NORMA kernels the netname service is part of the netmessage server. In NORMA the netname service is a modified version of *snames* which offers the old netname server interface.

with the server. The context port and the *BGN* are registered in the storage server's object location table. If the context exhausts the  $2^{16}$  LLI the proxy layer requests a new *BGN*.

The proxy layer requests the appropriate server to map a cluster as a consequence of an object fault on the cluster's head. When a storage server receives a map request, it searches its object location table to find if the cluster was mapped. If the cluster was not mapped, the server reads the cluster from disk and passes it to the context in an IPC message. The server also registers the object's LLI and the context's port in the object location table. If the cluster was mapped, the storage server returns a send right to the object's port and the LLI of its client proxy class. If this information is not present in the object location table, it is obtained from the context where the object was mapped. If needed the context will allocate a server proxy and a port for the object.

When a cluster is flushed or unmapped it is sent to the appropriate storage server in an IPC message. The server writes the new cluster state on disk. If the cluster is being unmapped its entry is removed from the object location table. Concurrent accesses to an object's persistent image are serialized using mutual exclusion primitives to synchronize the storage server's service threads.

## 4 Evaluation

This section evaluates some key aspects of MIKE's performance and compares MIKE with the versions of IK which only use traditional UNIX abstractions.

The test environment was composed of one 33 MHz i386 Intel 303 and a 33 MHz i486 Topzen, connected by a 10 Mbit *Ethernet*, running MK77<sup>3</sup> in the intra-node tests and NORMA 12<sup>4</sup>, in the inter-node tests. In the inter-node tests the client was in the i386 machine.

### 4.1 Object invocation performance

This section evaluates the performance of object invocation. The benchmark program is very simple – one object invokes an empty method on another object. The tables present the method signatures followed by the elapsed times. In the method signatures, **object\*** is a reference to a MIKE object. The values were determined by executing the test in a tight loop 100,000 times and computing the average elapsed time of each pass through the loop.

Table 1 compares MIKE's intra-context object invocation cost with a virtual method invocation in C++.

	MIKE	C++
<code>void aClass::f(int *)</code>	6.8 $\mu$ s	2.6 $\mu$ s

Table 1: Intra-context invocation elapsed times.

<sup>3</sup>Compiled with STD+WS.

<sup>4</sup>Compiled with STD+WS+assert+lineno+NORMA+norma\_ether.

C++ invocation is 62% faster for two reasons. Firstly, C++ uses dynamic binding, which is more efficient than the late binding used in MIKE. Secondly, C++'s binding mechanism is reentrant. MIKE uses a per-class method cache shared by all threads. Accesses to this cache are synchronized, adding approximately  $2\mu s$  to the invocation cost. A per-thread global cache could be more efficient, avoiding the need to synchronize accesses.

Table 2 presents remote invocation costs. Mapped means that the referenced object or an associated client proxy is mapped locally. With unmapped persistent object's references an optimization is used (see section 3.2.4).

		Intra-node	Inter-node
<code>void aClass::f(int *)</code>		315 $\mu s$	3053 $\mu s$
<code>void aClass::f(object *)</code>	unmapped	469 $\mu s$	3344 $\mu s$
<code>void aClass::f(object *)</code>	mapped	653 $\mu s$	3560 $\mu s$

Table 2: Remote invocation elapsed times.

Passing an object reference is more expensive than passing a pointer to an integer due to the costs of exporting and importing the reference. It is around 28% cheaper to export an unmapped object's reference because sending a port is relatively expensive.

The results expose the problem of performance transparency. Invocation costs grow significantly with the "distance" between the invoker and the invoked object. Therefore, the programmer (or the platform) should locate closely related objects together to achieve good performance.

Table 3 compares the time elapsed in a simple RPC, based directly in MiG stubs, with the time spent in invoking a method with similar parameters on a remote object.

<code>void f(int *)</code>	270 $\mu s$
<code>void aClass::f(int *)</code>	315 $\mu s$

Table 3: Intra-node remote invocation versus simply calling MiG stubs.

MIKE's remote invocation mechanism adds three object invocations to the MiG RPC code path – the invocation of the client proxy, the invocation of method `dispatch` in the server proxy and the invocation of the object. This combined with the overhead of dynamically managing the number of dispatcher threads adds 17% to the cost of a simple intra-node MiG RPC. We believe this extra cost is well justified by the functionality provided.

## 4.2 Implementation on UNIX vs. implementation on Mach

This section highlights the major benefits of using Mach to support a distributed object-oriented programming platform. It compares MIKE with the versions of IK that only use traditional UNIX abstractions.

In IK threads are implemented by user level code without kernel support, which has the usual advantages and disadvantages [14]. Since IK controls the thread scheduling,



preemption can only occur in well known points. This renders synchronization unnecessary in most accesses to shared data structures. In particular, there is no need to synchronize accesses to the method's cache, which makes IK's intra-context invocation faster than MIKE's.

One of the problems with IK's threads is the support for blocking system calls. IK redefines the most common blocking system calls to implement preemption. This mechanism is based on the SIGIO signal and the `select` system call and it introduces a significant performance overhead. It is responsible for 30% of the total cost of an inter-node remote invocation between a Sparc 10/20 and a Sparc 10/30, where the SunOS 4.1.3 LWP thread package is used [13]. Furthermore, when a thread blocks in a page fault requiring I/O, the entire context blocks.

On the other hand, MIKE uses Cthreads supported by multiple kernel threads. These threads can handle blocking operations more efficiently, including page faults. Mach threads support parallelism inside a context, while IK must use several contexts to explore the parallelism offered by a shared memory multiprocessor.

IK uses BSD sockets to perform remote invocations, this ensures portability and interoperability, but MIKE's remote invocation is faster than IK's. The main reasons for this are the more efficient IPC and threads implementation on Mach. In particular, MIKE's intra-node remote invocation is ten times faster than IK's SunOS version on top of a Sparc 10/30 [13] and the inter-node remote invocation is 30% faster than in IK (in the same test configuration as described in the previous paragraphs). On the other hand, inter-node remote invocation supported by the netmessage server is very slow.

The use of Mach ports as distributed object references played a fundamental role in the implementation of MIKE's distributed garbage collector and child death detection mechanisms. Another important advantage of using Mach ports is the per-object protection by capabilities.

In IK the name service and storage system code is linked with the run time support library. IK relies on a distributed file system to name persistent objects and access their images. MIKE takes advantage of the efficient Mach IPC and uses a multi-server architecture. This eliminates the dependency on a distributed file system to access persistent objects. Furthermore, the multi-server architecture will hopefully ease MIKE's port to other operating system "personalities", because its run time support uses almost only Mach abstractions and UNIX dependencies are isolated in the storage and name servers.

## 5 Related work

This section compares MIKE with other systems that exploit micro-kernel functionality with similar goals. Namely COOL [2], the multi-server [6] and Casper [15]. The first one exploits the Chorus [10] functionality and the last two exploit the facilities offered by Mach.

MIKE's distributed object sharing support is similar to the one in the multi-server. The two systems use a function shipping model implemented using Mach inter-process communication. Casper and COOL use data-shipping to share objects between contexts. Both systems use a page-based distributed shared memory similar to Li's [7], implemented with external pagers. COOL also supports an RPC based object sharing mechanism.

MIKE does not use a page-based distributed shared memory because of the well known false sharing problem. In fact, our average object size, 48 bytes, is much smaller than the

unit of sharing, the 4 Kbytes page. We believe the unit of sharing must be the language level object or even fragments of it [4]. Casper and COOL use object clustering techniques to minimize the false sharing problem.

Another problem with Casper and COOL's distributed shared memory implementation is that they offer a sequentially consistent memory, using page invalidation to detect and classify memory accesses. This memory model is expensive and usually accesses must also be synchronized at an higher level.

Object naming in MIKE and the multi-server is similar. They both associate ports with distributed objects and use port capabilities as distributed object references. However, MIKE supports object persistency, which raises several problems (e.g. send rights must be complemented with persistent identifiers to reference potentially persistent objects).

Casper and COOL also support persistency, but they use virtual memory pointers as distributed and persistent object references. This approach eliminates the need for pointer swizzling and allows persistent objects to be loaded by the normal demand paging scheme. On the other hand, the systems must ensure that contexts use the same addresses for the same distributed objects and persistent object addresses must be allocated persistently. This can be expensive and in Casper it restricts persistent store size to the virtual address space size of the processors memory management unit (4 Gbyte), which is a severe restriction. With the advent of wide address space processors this restriction will become unimportant. COOL avoids this restriction by supporting a form of pointer swizzling.

Local garbage collection in MIKE is similar to the one in Casper, but the former recycles persistent objects. This task is simplified by the use of a single storage server. This centralized storage server prevents scalability. Like MIKE, the multi-server uses no-more-senders notifications to implement distributed garbage collection, but the multi-server also uses reference counting for local garbage collection.

The multi-server, COOL and MIKE use C++ to program applications. In Casper programmers use the Napier88 programming language. MIKE, COOL and Casper support distribution and persistency transparently, but in the multi-server the proxies must be hand coded and the programmer must specify the RPC messages using a subset of an IDL.

## 6 Conclusion

We described an object-oriented programming platform which supports transparent access to remote and persistent objects programmed in C++. This implementation runs on top of the Mach NORMA kernel and the BSD single server. It makes extensive use of Mach IPC and real multi-threading facilities. Since the run time support library is built using mostly the micro-kernel interface, the port to other operating system "personalities" should be easy.

The use of Mach IPC and threads was determinant to achieve good remote object invocation performance. Mach threads also allow MIKE to support parallelism inside a single address space.

The use of ports as distributed object references allows for an easy implementation of distributed garbage collection and child death detection mechanisms and provides per-object protection by capabilities. In spite of this advantages, the use of send rights as references to potentially persistent objects proved insufficient. To solve this problem and

retain the advantages of using ports we complement send rights with a persistent global unique identifier.

The desire to support fine grained objects led us to the use of lazy evaluation techniques to minimize the overhead of associating a port per distributed object and using two “communication classes” per each original class. Ports are allocated in a lazy way and proxy classes are also loaded and linked lazily.

The EC++ proxy layer on top of MiG makes remote invocation very flexible without degrading performance. Proxy classes are generated by a tool in a way transparent to the programmer. Nevertheless, the programmer can modify the generated code to meet specific application needs. This architecture also eases the interconnection of MIKE’s applications with existing MiG servers.

We believe that our platform eases the development of distributed applications, but there is still a lot of work to do in hiding the lack of performance transparency and the partial failures from cooperating application components. Therefore, we are currently investigating data shipping based distributed object sharing techniques combined with group communication tools.

## Acknowledgements

We would like to thank the members of our group at INESC, in alphabetical order, Adriano Couto, Paulo Ferreira, Paulo Guedes, Cristina Lopes, David Matos, João Pereira, José Pereira, Mario Reis, Manuel Sequeira, António Silva and André Zúquete for their work in the IK platform. Special thanks go to Paulo Guedes and Manuel Sequeira for their careful reading of this manuscript and for their many suggestions that led to its improvement.

## Availability

MIKE and the distributed ray-tracer are available on request from the authors. E-mail can be sent to `ik-staff@sabrina.inesc.pt`.

## References

- [1] Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young. Mach: A New Kernel Foundation for UNIX Development. In *Proceedings of Summer Usenix*, July 1986.
- [2] P. Amaral, C. Jacquemot, P. Jensen, R. Lea, and A. Mirowski. Transparent Object Migration in COOL-2. In *Proceedings of Workshop on Dynamic Object Placement and Load-Balancing in Parallel and Distributed Systems, ECOOP’92*, June 1992.
- [3] Richard P. Draves. A Revised IPC Interface. In *Proceedings of the USENIX Mach Conference*, October 1990.
- [4] Michael Feeley and Henry Levy. Distributed Shared Memory with Versioned Objects. In *Proceedings of OOPSLA’92*, pages 247–262, Vancouver, Canada, October 1992.
- [5] Paulo Ferreira. Reclaiming Storage in an Object Oriented Platform Supporting Extended C++ and Objective-C Applications. In *Proceedings of the International Workshop on Object Orientation in Operating Systems - IEEE*, Palo-Alto, October 1991.
- [6] Paulo Guedes and Daniel Julin. Writing a Client-Server Application in C++. In *Proceedings of the USENIX C++ conference*, August 1992.

- [7] K. Li and P. Hudak. Memory Coherence in Shared Virtual Memory Systems. *ACM Transactions on Computer Systems*, 17(4):321–359, November 1989.
- [8] Keith Loepere. *Mach 3 Server Writer's Guide*. Open Software Foundation, January 1992.
- [9] José Alves Marques and Paulo Guedes. Extending the Operating System to Support an Object-Oriented Environment. In *Proceedings of OOPSLA '89*, pages 113–122, New Orleans, Louisiana, October 1989.
- [10] Mark Rozier, Vadim Abrozimov, Francois Armand, Ivan Boule, Frédéric Hermann, Michel Gien, Mark Guillemont, Claude Kaiser, Pierre Léonard, Sylvain Langlois, and Willi Neuhauser. Chorus Distributed Operating System. In *Computing Systems*, volume 1, pages 304–370, December 1988.
- [11] Manuel Sequeira and José Alves Marques. Can C++ be Used for Programming Distributed and Persistent Objects? In *Proceedings of the International Workshop on Object Orientation in Operating Systems - IEEE*, Palo Alto, October 1991.
- [12] Marc Shapiro. Structure and Encapsulation in Distributed Systems: The Proxy Principle. In *The 6th International Conference on Distributed Computer Systems*, pages 198–204, Cambridge, Mass., 1986. IEEE.
- [13] Pedro Sousa, Manuel Sequeira, André Zúquete, Paulo Guedes, and José Alves Marques. The IK Distributed and Persistent Platform — Overview and Evaluation. Technical report, IN-ESC, February 1993.
- [14] Avadis Tevanian, Richard F. Rashid, David B. Golub, David L. Black, Eric Cooper, and Michael Young. Mach Threads and the Unix Kernel: The Battle for Control. Technical Report CMU-CS-87-149, Department of Computer Science, Carnegie Mellon University, August 1987.
- [15] F. Vaughan, T. Basso, A. Dearle, C. Marlin, and C. Barter. Casper: a Cached Architecture Supporting Persistence. *Computing Systems*, 5(3):337–359, 1992.

# Task Migration on the top of the Mach Microkernel\*

Dejan S. Milojević<sup>†</sup>, Wolfgang Zint,  
Andreas Dangel and Peter Giese

University of Kaiserslautern, Informatik, Geb. 36, Zim. 436  
Erwin-Schrödingerstraße, 6750 Kaiserslautern, Germany  
tel (49 631) 205 3293 fax (49 631) 205 3558  
e-mail: [dejan,zint,dangel,giese]@informatik.uni-kl.de

## Abstract

This paper presents initial results in the design and implementation of task migration on the top of the Mach  $\mu$ kernel. The presented work is part of a broader project concerning research on load distribution. Our task migration is implemented in user space in order to improve portability, maintainability and flexibility. At the same time we paid attention not to sacrifice performance, transparency and functionality. Although we have implemented task migration in user space, some modifications to the kernel were necessary. We have designed on the  $\mu$ kernel abstraction level, unaware of issues such as files, signals or other UNIXisms which were main obstacles to simple and transparent process migration so far. We expect to benefit much more when we start dealing with load distribution decisions. Our design allows us to make scheduling decisions entirely based on Mach virtual memory, interprocess communication and processor load.

## 1 Introduction

The field of Load Distribution (LD), encompassing load balancing, load sharing and related mechanisms, such as Task Migration (TM) and remote execution, has been explored for quite some time [Bokh79, Eage86, Milo91]. Many practical and theoretical results have evolved, but none has achieved wide acceptance. There have been many obstacles, such as a relatively old operating system design, artificial support for distributed computing by extending stand-alone operating systems in a network, the lack of distributed applications, dependence on the particular modified version of the underlying operating system, limited functionality and transparency etc. We believe that contemporary  $\mu$ kernels provide a convenient base for LD and particularly for TM. In a  $\mu$ kernel, basic abstractions are supported within the kernel, while other functionality is provided within user space. This allows access to data and functionality formerly hidden inside the kernel, and thereby improves opportunities for a user level TM implementation. This and other characteristics, such as network transparency, modern virtual memory design and modularity, that modern  $\mu$ kernels possess (particularly Mach [Blac92]), are a promising base for yet another effort in the field of TM. Therefore, we have set as our goal to demonstrate that the Mach  $\mu$ kernel is a suitable environment for the implementation of TM. Our implementation aims at a transparent and portable migration in user space (possibly with minor modifications to the kernel), without paying significant penalties in performance and functionality.

The paper describes two user-space TM servers, necessary modifications to the kernel and a preliminary performance evaluation. Besides serving as a base for future work, the goal of this phase was

\*Research is supported by DAAD, University of Kaiserslautern (Germany) and Institute "Mihajlo Pupin", Belgrade (Yugoslavia).

<sup>†</sup>Currently on a leave from Institute "Mihajlo Pupin", Belgrade, (Yugoslavia).

to provide insight into the complexities of TM implementation, level of transparency, functionality and performance penalties paid for the user level implementation.

The rest of the paper is organized in the following manner. In this section we present an overview of the field and the relevant Mach characteristics. The design of our task migration scheme is presented in section 2. The implementation is described in section 3 by surveying the modifications applied to Mach, and two versions of the TM server. In section 4, we present some preliminary performance measurements. Related research is described in section 5. Our conclusions and future research are presented in section 6.

## 1.1 Overview of the Field

The field of TM has often been surveyed and described, e.g. in [Gosc91, Arts89]. Here, we briefly mention the motivations for TM, its issues and important implementations.

Computing history has shown that technology continuously provides more resources, but also increases demand for these resources. No matter how much is provided, there will always be an application that needs more, be it computing power, primary memory, huge paging space or specialized hardware. One of the convenient ways to satisfy this need is TM. Other applications of TM include the areas of fault-tolerance, real-time, system administration etc. Mobile computers may become yet another important area to apply TM [Doug92].

One of the intriguing issues of TM is **transparency**. It implies that a task can transparently execute after its migration, without recognizing that it has been migrated, except possibly for the performance difference. **User v. kernel space implementation** concerns the tradeoff between efficiency and simplicity. A user space implementation, while sacrificing speed to some extent, can provide for a simple, robust solution [Arts89]. A kernel implementation may be necessary in the case of real-time requirements. **Residual dependency** is related to the amount of the task state that may be left at the originating or some other node. It is directly related to the reliability of task migration. Task migration should be recoverable, i.e., it should be possible to recreate a task in the case that its migration has failed. **Autonomy and security** are important with respect to the environment as well as for the migrated task itself. A task should in no way influence the integrity of the environment that it is migrated to. Reclaiming of resources, if requested, should happen in a limited time. Finally, **performance** consists of immediate transfer and freeze time (the time during task is suspended), as well as the transfer time of lazy migrated state, such as applied for "Copy-On-Reference" (COR).

The history of TM is relatively old. It runs from early multiprocessors and distributed systems [Bokh79, Mill81], to the modern distributed environments and massively-parallel-processors [Zajc93]. It is an area where implementation dominates. Many prototypes have been running, but only a few have emerged as working products, e.g. Locus [Walk89]. We have chosen to mention the following work. **MOS(IX)** [Bara85] was one of the first to demonstrate  $\mu$ kernel appropriateness for LD, splitting the operating system into two layers. **Accent** [Zaya87], the predecessor of Mach, was the first to introduce the COR technique. The COR technique allows for faster address space transfer and achieves performance improvement by lazy copy of pages, thereby avoiding unnecessary page transfers. The **V kernel** introduced a technique known as "precopying" of the task address space [Thei85]. Precopying significantly improves freeze time, the period while the task is suspended during migration. **Charlotte's** TM scheme [Arts89] extensively relies on the underlying operating system, its communication mechanisms and language. Interprocess communication has been modified in order to support transparent network communication. **Locus** [Walk89] is an exception to most TM implementations because it was the only one to achieve a product stage. It has been ported to the AIX operating system on the IBM 370 and PS/2 computers under the name of Transparent Computing Facility. Locus has achieved a high level of functionality and transparency, paying the cost in significant kernel modifications. It has been recently ported to OSF/1 operating system, running on top of Mach [Zajc93]. **Sprite** [Doug91], is one of the recent and successful process migration implementations. It will be covered in more detail in section 5.

## 1.2 Relevant Mach Characteristics

The purpose of this subsection is not to give a Mach description, but rather to mention particular Mach features relevant to our TM scheme. More details about Mach can be obtained in [Blac92, Golu90]. The following issues are relevant to our scheme:

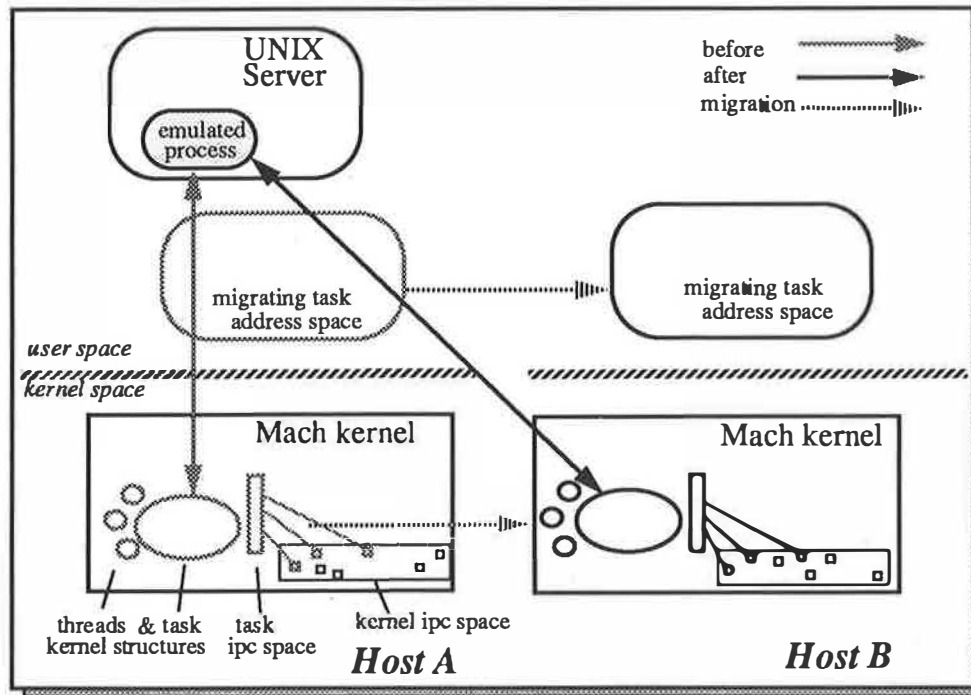


Figure 1: The transferred task state consists of the address space, IPC space, thread states, and other state, such as task and thread kernel ports, exception ports, task bootstrap port etc. The UNIX server state, contained within the task, is transferred implicitly, as a part of the address space (shared pages) and IPC space (rights through which the emulator communicates with the UNIX server).

- **Multicomputer support**, as provided by the Mach NORMA version, made our task much easier. The support for network Interprocess Communication (IPC) [Barr91], and Distributed Shared Memory (DSM) [Barr93] is of particular significance. Network IPC has been implemented within the kernel in order to improve performance. It transparently extends local IPC over the network. DSM overcomes current pagers' limitations to support only a single kernel. DSM code is transparently inserted between kernel and pager code, allowing existing pagers to support multiple kernels.
- **UNIX emulation** [Golu90] relieved us from handling UNIX abstractions in early design phases. However, this does not preclude us from considering modifications later, in the case of low performance. We have particularly observed the unacceptable performance of the emulator and the current implementation of mapped files, which have been optimized for parallel, but not for distributed, systems.
- **Device access**, based on IPC, provided a further level of transparency [Fori91]. Formerly, it used to be very hard, if not impossible, to extract the kernel state that remained in device drivers [Free91].

The Mach  $\mu$ kernel has the potential to become a standard basis for building operating systems. There are a lot of projects that have Mach as an underlying environment. Finally, its predecessor Accent was the base for TM experiments [Zaya87]. Mach inherited its design from Accent, particularly in some aspects of memory management. Therefore, Mach is a promising environment for TM experiments.

## 2 Task Migration Design

TM is implemented on top of Mach, but outside of any operating system emulation server. This implies that we migrate Mach tasks, while the UNIX process remains on the source machine, as presented in Figure 1. The obvious disadvantage is home dependency, a residual dependency with regard to the originating machine [Doug91]. The advantage is that the same TM scheme may be used for the applications

running on any operating system emulated on the top of Mach (e.g. BSD UNIX [Golu90], VMS [Wiec92] etc.), as well as for the applications running on the bare Mach. As in most TM implementations, our scheme consists of extracting the task state, transferring it across the network and reestablishing it at a new site. The world notices no difference while communicating with the migrated task, except for performance. The Mach task state consists of the address space, the capability space, the thread states and some other state. We are taking advantage of the Mach NORMA version which supports in-kernel network IPC and DSM. We do not implement capability migration and message forwarding; NORMA code takes care of that, as well as of DSM. However, not all of the task state can be transferred from within user space. We have faced the following problems in the design of TM.

- Mach objects are represented by kernel ports. Actions on behalf of the objects are performed by sending messages to their kernel ports. Therefore, it is necessary to preserve the task and thread kernel ports after migration. However, the kernel ports are not handled the same way as normal ports whose receive capabilities reside within the task IPC space. There are no receive capabilities for the kernel ports, and therefore there is no way to extract them using the existing Mach interface.
- Any shared or externally mapped memory causes a consistency problem for TM. Common examples are memory that is shared through inheritance and memory mapped files. Such memory areas are non-eligible for user space transfer with the current pagers, such as the default pager and the inode pager, which are designed to see only one kernel. The consistency problem could be overcome if Mach pagers would support multiple kernels. Basically, the same consistency support should be provided as in NORMA DSM. Internal, non-shared memory can simply be read from the source side and written into the task instance on the destination side. We call such areas “eligible” for user space transfer.

These were the only two cases where we did not have kernel support. As a solution to the first problem we provide two system calls to interpose the task and thread kernel ports. Both calls return the kernel port and leave the supplied port as a new kernel port. After that, we are the only ones who have the capability for the task kernel port and, therefore, only we can control it. All messages sent to a task end up at the extracted kernel port, whose receive capability is now in the TM server's IPC space. In the meantime, we perform actions on behalf of the task and transfer its state to the destination side. Once the state is transferred, we make the interpose call on the destination side, inserting the original task kernel port into the new task instance.

Shared memory and memory mapped files are a complex problem. However, this is not inherent in TM. Shared memory is not supported by most TM implementations, and files are handled by a distributed file system, e.g., in Sprite [Doug91], MOS(IX) [Bara85] and Locus [Walk89]. We adopt a similar approach. Shared memory and mapped files are supported by NORMA distributed shared memory. We currently handle it by exporting the pager port that represents a region of memory and then remapping it on the remote side. As soon as Mach exports the needed functionality, we may switch to it.

Except for the aforementioned two cases, everything else is implemented in user space. Our TM algorithm can be described with the following steps:

- suspend the task and abort the threads in order to clear the kernel state,
- interpose the task/thread kernel ports,
- transfer the address space, using various user and NORMA memory transfer strategies,
- transfer the threads by getting the state and setting it on the remote side,
- transfer the capabilities (we extract, transport and insert capabilities; NORMA does the actual port transfer),
- transfer the other task/thread state,
- interpose back the task/thread kernel ports at the destination site and
- resume the task.



Before migration, it is necessary to clear the thread kernel state, since it is almost impossible to transfer it. Typically, this case arises when threads are waiting within the kernel, e.g. for a message to be sent or received or a faulted page to be paged-in. Mach provides a particular call *thread\_abort*, which aborts thread execution within the kernel and leaves the thread at a clear point, containing no kernel state inconvenient for migration. The execution is restarted after migration, unless instructed otherwise. Lux et al. describe how kernel state complicates migration in BirliX [Lux92].

The adopted design relieves us from issues, such as signals and distributed file systems, which have traditionally been a nightmare in TM and remote execution implementations. We expect major benefits later, when we are to make LD decisions. We shall make decisions only in terms of Virtual Memory (VM and XMM), IPC and processor load, while previous systems separately dealt with files, network traffic, local and remote IPC. Of particular interest for us are XMM and network IPC which, as we presume, dominate the costs.

### 3 Implementation

The underlying environment in our TM research consists of 3 PCs, interconnected via Ethernet. The PCs are based on a 33 MHz i80486 processor with 8 MB RAM and 400MB SCSI disk. We have been using the Mach NORMA version (MK7, MK12, MK13) and the UNIX server UX28. TM has been accomplished by designing and implementing two versions of migration server, a simple and an optimized one. They have been implemented in a relatively short period. We installed our first PC at the end of November 1991, and the current environment at the end of March 1992. At the end of May we migrated a task for the first time. Polishing it up took more time, which is partially due to the parallel NORMA development. The task migration is currently stable for test applications.

The routines concerning the kernel modifications fit in a file of 300 lines of C code, most of which are comments, debugging code and assertions. The simple migration server has about 600 lines of code, and is actually a library routine. It could be used either by linking it to the task that is going to initiate migration, or by providing a server interface, in which case the migration code is linked to the server. In the former case it is not possible for a task to migrate itself, since one of the first actions on behalf of the migrating task is to suspend it, causing a deadlock. In such a case, it is necessary to provide a server which will indirectly start the migration. The optimized migration server has about 13000 lines. It is based on cooperating servers running on all nodes and has a few threads of control.

In the following subsections we describe the kernel modifications and the simple and optimized TM server.

#### 3.1 Necessary Modifications to the Mach Microkernel

As mentioned in the previous section, the kernel modifications consist of the kernel port interpose, which is a necessary, permanent modification, and a temporary extension to export the memory object port until the NORMA code exports the needed functionality.

**The kernel port interpose** modification introduced two new system calls, one for the task and one for the thread kernel port interposition. Like most other calls, these are actually messages, sent to the host on which the task executes. Unfortunately, it can not be sent to the task or thread. The interpose routine takes care that all messages accumulated in the original kernel port (while being interposed) are handled properly and the threads blocked in the full port message queue are woken up and their requests handled. In order to avoid unnecessary blocking, the queue limit for the interposed kernel ports is increased to its maximum. Once the ports are interposed back on the destination side, the default queue limit is restored. Interposing, as presented in Figure 2, consists of exchanging the pointer to the IPC space the port belongs to, as well as modifying some other internal task state. On the return from the call, the receive capability for the task kernel port resides within the server IPC space and the receive capability for the interposing port is extracted from the server IPC space into the kernel IPC space.

**Exporting the pager port** is a temporary modification. The existing NORMA version does not support distributed memory shared through inheritance, although DSM functionality exists. Therefore, we adopted a temporary solution. We export the memory object for the particular region. For security reasons, the Mach interface does not export the memory object port capability, but only the related

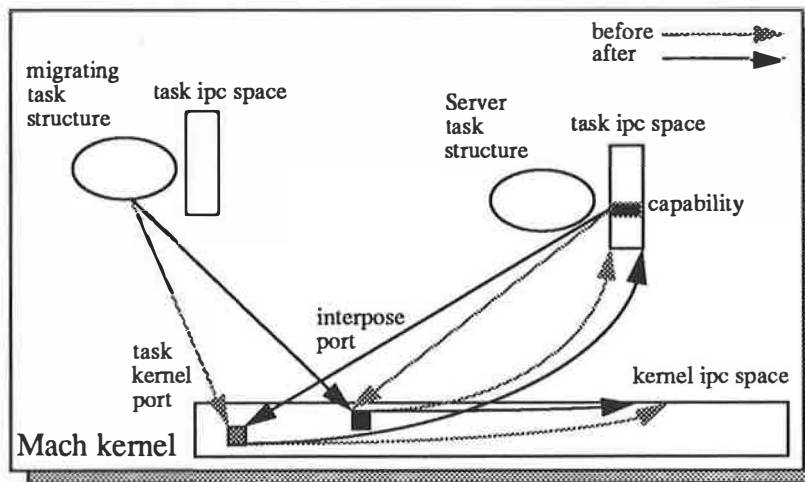


Figure 2: *Interposing the task kernel port - internal perspective: the task\_interpose routine exchanges the port that represents the task with an interposing port; it prerequisites exchanging pointers to ports and the IPC spaces the ports belong to, namely kernel IPC space with the IPC space of the task that initiates migration.*

name capability [Youn89]. We extract the capability for the memory object port and map it onto the remote side. This provided the necessary functionality in the early design phases. The added system call looks up the memory object port which represents the memory region and exports a send right out of the kernel.

### 3.2 Simple Migration Server

At the very beginning the Simple Migration Server (SMS) was implemented in order to verify design feasibility and to get the initial insight into performance. Later on, we completely switched to it. It relies entirely on the NORMA memory transfer and is deliberately unoptimized. From the memory transfer point of view, the SMS has better performance than the Optimized Migration Server (OMS), which relies on user level address space transfer and is more flexible regarding the choice of the memory transfer strategy. Regarding capabilities, threads and other state transfer, the SMS has worse performance than the OMS, since each capability, thread state, etc., is transferred separately, resulting in a new message sent across the network. The SMS consists of a few parts that migrate capabilities, memory, thread state and the other task state. Before migration, the task is suspended and its kernel port is interposed. From now on, we are sure that no unprivileged user has access to the task. A privileged user can always obtain a send capability for the current task kernel port, regardless of the fact that we have exchanged it.

The **thread migration** consists of transferring the thread state. There are a few different flavors of the thread state which should be transferred, such as the contents of CPU and FPU registers etc. The thread states are extracted on the source side and inserted into the newly created threads on the destination side. We have never had an interest in any particular state *per se*, we just copied all of them. This improves the portability of our TM scheme across various processor architectures.

The **memory migration** is based on the default NORMA COR strategy, as opposed to the optimized server where different strategies are implemented in user space. We wanted to explore different approaches in order to get more insight into performance and functionality tradeoffs. In both approaches, the task address space is analyzed, and areas not eligible for user-space migration are transferred using NORMA, as discussed in section 2. These areas are mapped in the destination task, using NORMA DSM. The rest of the memory is either copied in user space (OMS) or using NORMA support (SMS). In either case, non-initialized areas are simply re-allocated.

The **capability migration** is performed by extracting the capabilities on the source node and inserting them on the destination node. We migrate send, send-once and receive capabilities. Port sets are

migrated by extracting all receive capabilities from the port set, migrating them and reconstructing the port set on the remote side. Capabilities are migrated one by one in SMS. This has detrimental effects on performance, due to the high costs involved in capability migration. Therefore, we may switch to the same method as in the OMS: extract all capabilities, transfer them in a message to the destination node, and insert them. This requires servers on both sides.

The other state consists of suspend counts for task and threads, bootstrap ports, exception ports etc. Each of these states is appropriately extracted and later inserted into the migrated instance of the task. Some of the state is transferred as part of creating the remote instance of the migrated task.

### 3.3 Optimized Migration Server

The Optimized Migration Server (OMS) has functionality similar to SMS, except that it supports user-space memory transfer and various optimizations. Only memory eligible for user-space migration is transferred across the network by the OMS, while shared and externally mapped memory is still transferred by NORMA. The OMS treats error recovery in the case of an erroneous migration, and potentially retries migration on the same or a different destination node. The following data transfer strategies are supported:

- **Flushing residual dependency** causes the transfer of all memory that remained either on the source or on other nodes due to the previous migrations.
- **Precopy** is implemented similarly to the V kernel; however, its performance rendered it useless: it was too slow to switch from the default pager to a new pager. The basic idea is to remap the address space to the new pager and then to extract the information about modified pages. We are investigating improvements.
- **Copy on reference** is supported in the classical way, similar to Accent or NORMA, but in user space. There is also an extended version that transfers a few pages surrounding the faulting page.
- **Read ahead** is meant to transfer the whole task address space within a given time. It periodically transfers some of the remaining pages to the destination node, until all pages are transferred.
- **Direct copy** is a simple transfer of all memory from the source node to the destination node.

The applied optimizations consist of **packing all capabilities and other relevant information in messages**, requiring servers on both machines; **overlapping of various state transfer**, e.g. memory with capability and thread transfer; **truncating zero ended memory areas** and **substitution of emulator and emulation vectors** by local instances. OMS uses an External Memory Manager (EMM) to provide the functionality for user-level memory transfer strategies. The address space of the migrating task is reconstructed on the destination side by remapping memory objects. After that, the EMM receives paging requests, and either transports referenced pages over the network or retrieves them from the local paging-file. The basic COR-mechanism is modified for COR-extend to fetch some remote pages surrounding the faulting page and transport the requested pages in parallel. Extra pages are saved locally to resolve succeeding page faults. Flushing the entire address space to backing store, as done in Sprite, doesn't make much sense in our environment with local paging files. Hence OMS uses a variant of the strategy. It transfers all eligible pages in parallel with the thread state and capability migration. Network and paging file access are serviced by separate threads to increase performance. The paging file is accessed by a stand-alone file system (the same as the default memory manager uses) to avoid deadlock with the BSD server.

OMS has been useful during development, however, despite many opportunities, currently we have abandoned the use of OMS due to the following reasons:

- Task memory regions not eligible for user-space transfer, such as shared memory or memory backed up by specific pagers, such as the inode pager, still need support of XMM code or the support of new distributed pagers. We have considered the latter case, but we found that it has not been preferred by other main Mach developers. Choosing this way would mean a departure from the mainstream development. In the former case, we duplicate the functionality.

- We have found SMS more robust for further research on LD. Its simplicity, despite somewhat slower performance, e.g. for port transfer, allows us to trace the problems more easily. OMS requires few threads of control, a separate pager, etc., making debugging harder.
- Our main goal was not TM itself, but rather LD, where TM is one of its important mechanisms. Migration candidates are usually long lived tasks (range of tenths of seconds); therefore the performance difference of a few hundred milliseconds between SMS and OMS is of secondary importance.

If we later, for any reason, find SMS inappropriate, we can always re-investigate the choice of migration techniques once again. One of the most likely reasons for reviving the OMS would be the comparison of various migration strategies. This would become an issue once we have a working LD scheme and other Mach applications, such as a distributed file system. As a short term improvement, we may merge some of the good characteristics of OMS, such as packing rights and other state, in a few messages, avoiding many short messages over the network.

## 4 Performance Measurements

Performance measurements are an important part of any TM implementation. Unfortunately it is hard to find adequate applications for measurements. There are a few true distributed applications, particularly for the Mach  $\mu$ kernel. Most researchers in the field of LD have made artificial loads or used some benchmarks; they rarely used real applications. We used the following: Artificial Load Task (ALT), WPI benchmark suite, parallel make support; matrix multiplication, simulation programs, and some other applications.

All results presented should be accepted with caution. There are many influences on its accuracy, since TM and the Mach NORMA version are continuously being changed. We present performance measurements for the sake of completeness and to give more insight into the order of magnitude. All results presented in Figures 3 to 10 are obtained as a mean of the five consecutive measurements. Only the results for Figures 11 and 12 are the mean of two out of the five measurements, since the experiment failed for some input values due to a known bug. If not otherwise indicated, SMS has been used. All measurements have been performed on norma13<sup>1</sup> and [UX,emulator]28.

### 4.1 Migration Server Measurements

In this subsection we present some low-level measurement results for both migration servers. In order to gain insight into the influence of particular migration parameters on the overall performance, we designed an Artificial Load Task (ALT). We can tune the following ALT parameters: ratios between computation, IPC activity and memory access, amount of memory (internally mapped by the default memory manager, locally and remotely shared memory), the number of threads, and the number of capabilities (receive, send for local and remote ports). ALT loops for a given number of iterations and in each loop it performs a number of remote procedure calls to the local and remote server, accesses locally and remotely mapped shared memory, followed by a computation part, currently represented by the Linpack benchmark. The main idea behind ALT is to represent the task VM, IPC and processor load behavior. We plan to experiment with various applications and thereby obtain realistic values for ALT parameters and use it as an artificial load for distributed scheduling experiments.

TM consists of three phases which transfer virtual memory, threads and capabilities. Each of these phases was measured using ALT. In Figure 3, we present measurements of transfer time v. memory size, performed using SMS, i.e. using NORMA default COR strategy. Transfer time as a function of the thread number is presented in Figure 4. Figure 5 shows transfer time as a function of the number

<sup>1</sup>In the meanwhile we have also ported SMS to norma14, however, we shall still present results only for norma13 version for a few reasons. First, although norma14 has been more stable for many previous problems, we did observe some new ones, preventing stable measurements. Second, we have switched within SMS to a new kernel supported address space transfer. This could favor SMS over OMS, which hasn't been ported to norma14. Finally, except for XMM, we have not observed any significant difference in relative performance behavior between the two versions. Despite some changes, the curves presented in Appendix A remained similar. We observed faster address space transfer (above 30%), but slower lazy copying.

of receive capabilities. Comparison of receive and send capabilities transfer is given in Figure 6. In Figure 7, we present an interesting side-effect of extending a VM layer to a distributed environment. While experimenting with memory transfers, we noticed unacceptable values for a particular test task. The test task address space regions are created by its parent using a *vm.write* system call on a page-by-page basis. As we closely inspected the address space, we saw that it consisted of many entries, which could be, but have not been, compacted into one. In local cases this does not represent a bottleneck, however, once that task address space is being transferred across the network, entry transfer cost is significantly increased. As we modified the program slightly to write all the memory in a big chunk instead of page-by-page, thereby reducing the number of entries, we significantly improved transfer time. This anomaly could be easily overcome by using the existing kernel function *vm\_map\_simplify*. From aforementioned measurements we can conclude that migration transfer time is independent of task address space size. It is a linear function of the number of internal memory regions, but they are usually limited in number, except for the unoptimized cases, such as the one presented in Figure 7. The number of threads and rights are usually small for tasks representing UNIX process, except for the servers, which are not good candidates for migrations, anyway. Except for address space transfer which is characterized by fluctuations, all curves are linear. Address space transfer has been subject to change with various norma versions, therefore we didn't further inspect the reasons for its fluctuation.

The presented measurements characterize the transfer phase. Equally important is performance during the task execution after migration, which depends on the transfer of residual state. Figure 8 presents an interesting and unexpected behavior of COR for OMS and SMS. While we expected better capability transfer performance for OMS, we did not expect better address space transfer. According to the measurements, it seems like OMS also has better address space transfer. The measurements are performed with an ALT version in which only the first integer of the page is written. Since OMS introduces an optimization of transferring the data up to the last non-null character in an area, just one integer is transferred instead of the page. As soon as the last integer within the page is written, performance of OMS decreased below the SMS performance. Performance of SMS is the same in both cases, since it has page granularity. This simple experiment demonstrates the benefits of implementing user-space TM, since it is rather easy to substitute various strategies and parameters. In Figure 9, we demonstrate the expected difference in transferring receive capabilities for SMS and OMS. The performance for send capabilities is similar and is not presented. Finally, the obvious benefit of migration towards a server with which the task communicates is demonstrated in Figure 10, where we present the task execution time as a function of the number of remote procedure calls with a server to/from which we migrate the task.

Based on above results we can conclude that the performance of our TM implementation is comparable to other kernel supported implementations, while it significantly outperforms user space implementations. For example, reported process migration in Sprite [Doug91] is few hundred milliseconds for a standard process. We have measured similar performance for our TM implementation<sup>2</sup>. In Condor [Litz92], it takes two minutes to migrate a 6MB process. In our servers it wouldn't cost much more to migrate any other average size task (assuming that memory regions are not chopped in many entries, which is not likely to be). Of course, the actual transfer is done lazy, preventing direct comparison. Similar results of user level implementations have been reported by other implementors, e.g. [Alon88]. It should be noted, though, that we have not aimed at optimizing performance in this phase of the project. For completeness, we shall compare our implementation with the performance of other systems. We derived formulas similar to those that the implementors of Charlotte have presented<sup>3</sup>.

Accent time (ms) =  $1180 + 115*s$   
V time (ms) =  $80 + 6*s$   
Charlotte time (ms) =  $45 + 78*p + 12.2*s + 9.9*r + 1.7*q$   
Sprite time (ms) =  $76 + 9.4*f + 0.48*fs + 0.66*s$   
SMS time (ms)  $\approx 150 + 48*n + 22.8*r + 5.5*s + 5.5*so + 58*t$   
OMS time (ms)  $\approx 50 + 2.4*n + 7.9*r + 1.9*s + 1.1*so + 5.4*t$

In the above formulas the following parameters have been used. In Charlotte: *p* is process migration management parameter with values [0,1,0.2], *s* is the virtual memory size in 2kB blocks, *r* is 0 if links are

<sup>2</sup>Our measurements are related to task, not process, migration, therefore, additional time should be accounted for. The same is true for most other examples.

<sup>3</sup>Results on Accent, V and Charlotte origin from [Arts89] and for Sprite from [Doug91], since they are more recent.

local, 1 otherwise, and  $q$  is the number of nonlocal links. In V, Sprite and Accent,  $s$  is VM size in kB,  $f$  is the number of open files and  $fs$  is file size in kB. In SMS and OMS,  $r$  is the number of receive rights,  $s$  is the number of send rights,  $so$  is the number of send-once rights, and  $n$  is the number of regions. According to these formulas, a typical UNIX process (or the corresponding task in relevant cases) would migrate in 330ms in Sprite (exec-time process), 680ms in V, 13 sec in Accent, 750ms in Charlotte, 500ms in SMS and 250ms in OMS. It should be noted that the presented performance measurements have been performed on different computers, therefore direct comparison is inappropriate.

## 4.2 WPI Benchmarks

We obtained interesting results while running the Jigsaw benchmark [Fink90]. The Jigsaw benchmark stresses the memory management activities of the operating system. It consists of making, scrambling, and solving a puzzle. During each phase, there is an allocation of huge amounts of memory. We repeat similar experiments enhanced by TM in four cases. The first case consists of locally executed Jigsaw. In the second case, we migrate the task after some amount of memory allocation, and thereby distribute memory on both machines. In the third case, the task is migrated after all memory is allocated on the source node. Finally, in the fourth case, the task is executed remotely, resulting in memory allocation mostly on the destination node. These four cases are presented in Figures 11 and 12. The worst performance is achieved in the third case where the task has to access all of its memory remotely. The first and the fourth case have similar performance, since all of the task's memory is on the node where the task executes. The second case exhibits the best performance for the tile sizes which cause excessive paging. In this case, the task has memory on both machines, so it starts thrashing later and achieves better performance. From this example we do not benefit too much, unless we apply similar reasoning as in memory overloaded systems. Distributing the memory across the network could improve performance in much the same way as distributing the load in an overloaded machine.

Finally, in Figure 13 we present the difference between the execution of a real make program and the WPI synthetic gcc compilation, *scomp*. For the real make, there is a performance penalty of more than 500% for the remote execution, while the *scomp* exhibits about a 50% degradation. This simple example demonstrates the inappropriateness of the existing emulator and the current implementation of a mapped file system for TM. The emulator has been designed to support UNIX emulation in a non-distributed environment. The task communicates through the emulator with the server by using shared memory and IPC. In the local case using shared memory for emulator/server communication does not impose a significant overhead. However, it turns out to be inappropriate for a distributed system, due to the unnecessary transfers of a complete page across the network, even for small information exchange. Emulator issues have also been investigated in [Pati93]. Similar reasoning is valid for mapped files. It is not the technique of memory mapped files itself that is a bottleneck; rather it is the current implementation which needs to be optimized for a distributed environment. We expect significant improvements when the existing emulator is replaced and file access optimized. This is, however, related to the particular operating system personality. Therefore, we plan to upgrade to OSF/1 server with a new distributed file system that is currently being developed within the OSF cluster project [Roga92].

## 4.3 Parallel Make and other Applications

One of the traditional TM applications is a parallel make. We are using *pmake*, written by Adam de Boor for Sprite. We modified it by inserting our migration routines. Due to the emulator problems, we are still not using it extensively. We expect major results to evolve when we switch to a new distributed file system. With current file system support, performance is significantly degraded. We are also running other applications, such as matrix multiplication, simulations, various benchmarks, etc. Most of them have been ported straightforwardly, e.g. simulations, which have been used as common UNIX programs. For matrix multiplication and the traveling salesman problem we have put additional effort to port them to Mach instead of UNIX. These programs run smoothly for small problem sizes, however, for big problem sizes we still encounter problems, which are likely to stem from some current limitations in Norma versions. We expect to have more results on these applications once we start making LD decisions.

## 5 Related Research

There is a lot of related research in the area of TM and LD. We have chosen some that represents TM on  $\mu$ kernels, monolithic kernels and in user-space.

- **Chorus** was expanded to support TM [Phil92], as a basis for load balancing experiments. The TM consists of elements similar to our scheme, but is applied on the process, instead of the Actor, level (Actors correspond to Mach tasks). It is more oriented towards a hypercube implementation. Some limitations arise from the fact that Chorus currently doesn't support port migration. It is too early, though, to draw deeper comparison with our research, since both projects are at the early stage of development.
- Closely related to our research are University of Utah - "**Schizo**" project [Swan93] and the **TCN** project, performed by Locus for Intel Paragon and OSF/1 AD [Zajc93]. Both projects are Mach based. "Schizo" is targeted at a distributed environment consisting of autonomous workstations. It investigates issues such as autonomy and privacy. "Schizo" inherited some aspects of the Stealth project [Krue91], namely prioritizing VM, file and CPU scheduling, extending it with prioritizing IPC, more robust failure handling, etc. We consider a cluster environment in our work, and therefore do not concern such issues. Our work on task migration has been used as one of the mechanisms for the "Schizo" project. The **TCN** project is concerned with traditional issues of monolithic systems, providing process migration and explicitly considering files, signals and sockets. Currently, TCN is only partially concerned with  $\mu$ kernel TM issues. In the OSF/1 environment, the Mach interface is not exposed to the user, and therefore atomicity of process migration is not affected. A complete solution would be a combination of our work and the work on TCN, i.e. a process migration that makes use of task migration.
- **Sprite** supports one of the important TM implementations [Doug91]. It is tightly coupled with the distributed file system. The address space transfer is optimized by using various caching techniques and by flushing the state onto the server. Compared to Sprite, we have acted on the lower level. We have never dealt with issues such as files or signals. In our scheme, these abstractions are transparently supported by the network IPC and DSM, which correspond to handling open I/O streams and caching in Sprite [Welc92, Doug92]. Therefore, there is an illusion that we have achieved a simpler design by retaining the existing file system instead of introducing the distributed one. However, for performance reasons, the current implementation of signals, files etc., needs to be optimized for distributed systems. This is, however, the flavor of the operating system personality emulated on top of Mach and is related neither to Mach nor to our scheme. It will be hard to compete with Sprite in performance, since their approach allowed for various optimizations regarding caching on the server and client side. Sprite is not only a challenge to our TM, but to Mach itself. All  $\mu$ kernel advantages, such as portability, flexibility, maintainability etc., should be confirmed with performance comparable to monolithic kernels, such as Sprite [Welc91]. Sprite has recently been ported to Mach [Kupf93], unfortunately process migration is not supported, preventing us from interesting comparison.
- **Condor** TM [Litz92] is dedicated to long running jobs that do not need transparency and allow for limitations on system calls the migrated task may issue. Performance penalties in Condor are much higher because it is necessary to dump the core, combine it with executables and return it to the submitting machine. The Condor approach is acceptable for long-running, computation-intensive jobs; for other classes of jobs, it is too expensive. Due to the less expensive migration, our scheme pays off for short running tasks.

## 6 Conclusion

We have presented a TM design and implementation on top of the Mach  $\mu$ kernel. We explained the reasons why we believe that our choice of a state-of-the-art message passing  $\mu$ kernel for TM implementation is at least comparable to other platforms, while in some aspects it has advantages. We have described minor

modifications we had to apply to the Mach  $\mu$ kernel in order to transparently migrate tasks, and two implementations of migration server. Afterwards, we presented some measurements of TM performance, as well as performance of the migrated applications. Finally, we compared our research to the related projects in this area.

As we base our research on Mach, we have tried to use existing Mach features as much as possible. Most of the issues, e.g. network IPC, DSM, and exporting the task/kernel state, have already been provided by Mach.

The work we presented in this paper is a part of wider research on load distribution. Task migration is a problem that we have to solve efficiently, but we expect to achieve major contributions later, when we start using task migration in the framework of load distribution. In the present phase of our work we can summarize the following contributions.

- We showed that it is possible to implement TM on top of the Mach  $\mu$ kernel without much effort, and that  $\mu$ kernels are a suitable level to implement TM on. We have acted in user space with minimal modifications to the kernel. The user space implementation significantly improved maintainability and extensibility.
- Our implementation has achieved high transparency without paying the price in performance comparable to other user space implementations, such as Condor. TM is completely transparent to the migrating task and to the tasks it communicates with. There is no need to link the task with any special libraries, nor are there any limitations to the calls that the task may issue. Since we do not depend on hardware, TM is also portable across the various hardware architectures. We act on the Mach level, transparently supporting various operating system emulations, e.g. UNIX or VMS, as well as applications running on bare Mach.
- We supported memory transfer strategies, such as precopy, flushing and COR in user space. Formerly, only a single strategy per TM implementation has been provided, implemented in the kernel.

OSF Grenoble Research Institute and the University of Utah are currently investigating our TM scheme as a possible technology for use in their "OSF/1 Cluster" and "Schizo" projects respectively. A collaboration with the OSF Grenoble Research Institute is underway, which should involve continuation of our work, namely load information and distributed scheduling.

Our further research is related to TM improvement, as a short-term goal, and to LD, as a long-term goal. In the first phase of TM we have provided functionality; now it is necessary to improve performance. We are going to combine the migration servers into one unique server that will merge the best characteristics of both. Then we need to modify or completely eliminate the emulator, since it has been designed for local execution and as such is not acceptable for distributed systems. It would also be necessary to redesign the mapped file system in order to improve performance. We shall continue to profile various applications and try to learn more about the task characteristics as a function of IPC, VM and processor load. In particular, we shall stress our experiments on IPC and VM dependency. Processor load behavior has been much more researched and existing results could be readily applied, while VM and IPC are less investigated. As a part of this research, we are currently implementing a load information management scheme for Mach and a user level IPC profiler.

As a long-term goal, we have started to work on distributed scheduling. We are interested in the relation of global and local scheduling as suggested by Krueger [Krue87]. Krueger shows that not every local scheduling algorithm matches the distributed one and vice versa. In traditional systems it was hard to access and modify local schedulers, while Mach may be suitable for such experiments [Blac90].

## 7 Availability

All mentioned programs are available upon request. They have still not been put for anonymous ftp access due to the continuous adjustments to new norma versions. It should be noted that OMS server is not supported anymore, although the last version is available. Beside TM related programs, we have available preliminary versions of load information manager (including user space IPC profiler) and simple distributed scheduler, supporting few basic strategies. Complete LD scheme with numerous strategies is expected to be finished in the second half of 1993.



## 8 Acknowledgements

We would like to thank: CMU and OSF people for providing us with Mach and with continuous support; Joe Barrera for important hints during development; Fred Dougliis and Brent Welch for discussing Mach and Sprite peculiarities; Prof. Dušan Velašević for initial encouragement and support; Prof. Amnon Barak and Prof. Andzej Goscinski for continuous support; Prof. Nehmer for making all of this happen. Special thanks are due to Fred Dougliis, Mike Kupfer, Jelena Vučetić, Nikola Šerbedžija, Sean O'Neill and Holger Assenmacher for proofreading the paper and giving many useful suggestions. Finally we would like to thank the program committee, the chairman David Black, and our liaison to program committee, Jay Lepreau, for many useful suggestions and for giving us the opportunity to present our work.

## References

- [Alon88] ALONSO, R. and KYRIMIS, K. (February 1988) *A Process Migration Implementation for a Unix System*. Proceedings of the USENIX Winter Conference, pages 365–372.
- [Arts89] ARTSY, Y. and FINKEL, R. (September 1989) *Designing a Process Migration Facility: The Charlotte Experience*. Computer, pages 47–56.
- [Bara85] BARAK, A. and SHILOH, A. (September 1985) *A Distributed Load-Balancing Policy for a Multicomputer*. Software-Practice and Experience, 15:901–913.
- [Barr91] BARRERA, J. (November 1991) *A Fast Mach Network IPC Implementation*. Proceedings of the Second USENIX Mach Symposium, pages 1–12.
- [Barr93] BARRERA, J. (1993) *Operating System Support for Multicomputers*. Technical Report CMU-CS-93-, PhD Thesis in Preparation.
- [Blac90] BLACK, D. (October 1990) *The Mach Timing Facility: An Implementation of Accurate Low-Overhead Usage Timing*. Proceedings of the USENIX Mach Workshop, pages 53–73.
- [Blac92] BLACK, L. D., ET AL (1992) *Microkernel Operating System Architecture and Mach*. Workshop on Micro-Kernels and Other Kernel Architectures, pages 11–30.
- [Bokh79] BOKHARI, S. H. (July 1979) *Dual Processor Scheduling with Dynamic Reassignment*. Trans. on SE, SE-5:326–334.
- [Doug91] DOUGLIS, F. and OUSTERHOUT, J. (August 1991) *Transparent Process Migration: Design Alternatives and the Sprite Implementation*. Software-Practice and Experience, 21:757–785.
- [Doug92] DOUGLIS, F. Personal communication 1992.
- [Eage86] EAGER, D., LAZOWSKA, E., and ZAHORJAN, J. (May 1986) *Dynamic Load Sharing in Homogeneous Distributed Systems*. Trans. on , SE-12:662–675.
- [Fink90] FINKEL, D., KINICKI, R. E., AJU, J., NICHOLS, B., and RAO, S. (October 1990) *Developing Benchmarks to Measure the performance of the Mach Operating System*. Proceedings of the USENIX Mach Workshop, pages 93–100.
- [Fori91] FORIN, A., GOLUB, D., and BERSHAD, B. (November 1991) *An I/O System for Mach 3.0*. Proceedings of the Second USENIX Mach Symposium, pages 163–176.
- [Free91] FREEDMAN, D. (January, 1991) *Experience Building a Process Migration Subsystem for UNIX*. Winter USENIX Conference, pages 349–355.
- [Golu90] GOLUB, D., DEAN, R., FORIN, A., and RASHID, R. (June 1990) *UNIX as an Application Program*. Proceedings of the Summer USENIX Conference, pages 87–95.
- [Gosc91] GOSCINSKI, A. (1991) *Distributed Operating Systems The Logical Design*. Addison-Wesley, pages 399–432.

- [Krue87] KRUEGER, P. and LIVNY, M. (August 1987) *Load balancing, load sharing and performance in distributed systems*. Technical Report 700, CS Department, University of Wisconsin-Madison.
- [Krue91] KRUEGER, P. and CHAWLA, R. (June 1991) *The Stealth Distributed Scheduler*. Proceedings of the 11th International Conference on Distributed Computing Systems, pages 336-343.
- [Kupf93] KUPFER, M. (April 1993) *Sprite on Mach*. Third USENIX Mach Symposium, this issue.
- [Litz92] LITZKOW, M. and SOLOMON, M. (January 1992) *Supporting Checkpointing and Process Migration outside the UNIX Kernel*. Proceedings of the USENIX Winter Conference, pages 283-290.
- [Lux92] LUX, W., KUHNHAUSER, W. E., and HARTIG, H. (June 1992) *The BirliX Migration Mechanism*. Position paper, presented at the Workshop on Dynamic Object Placement and Load Balancing in Parallel and Distributed Systems Programme, pages 83-90.
- [Mill81] MILLER, B. and PRESOTTO, D. (1981) *XOS: an Operating System for the XTREE Architecture*. Operating Systems Review, 2:21-32.
- [Milo91] MILOJICIC, D., PJEVAC, M., and VELASEVIC, D. (September 1991) *Load Balancing Survey*. Proceedings of the Summer EurOpen Conference, pages 157-172.
- [Pati93] PATIENCE, S. (April 1993) *Redirecting System Calls in Mach 3.0: An alternative to the emulator*. Third USENIX Mach Symposium, this issue.
- [Phil92] PHILIPPE, L. (1992) *La Migration de Processus*. Technical Report Chorus Systèmes, Phd Thesis in preparation.
- [Roga92] ROGADO, J. (October 1992) *A Strawman Proposal for the Cluster Project*. Technical Report OSF RI, Grenoble.
- [Swan93] SWANSON, M., STOLLER, L., CRITCHLOW, T., and KESSLER, R. (April 1993) *The Design of the Schizophrenic Workstation System*. Third USENIX Mach Symposium, this issue.
- [Thei85] THEIMER, M., LANTZ, K., and CHERITON, D. (1985) *Preemptable Remote Execution Facilities for the V System*. Proceedings of the 10th ACM Symposium on OS Principles, pages 2-12.
- [Walk89] WALKER, B. J. and MATHEWS, R. M. (1989) *Process Migration in AIX's Transparent Computing Facility*. IEEE Technical Committee on Operating Systems, 3:5-7.
- [Welc91] WELCH, B. (November 1991) *The File System Belongs in the Kernel*. Proceedings of the Second USENIX Mach Symposium, pages 233-250.
- [Welc92] WELCH, B. Personal communication 1992.
- [Wiec92] WIECEK, C. A. (April 1992) *A Model and Prototype of VMS Using the Mach 3.0 Kernel*. Workshop on Micro-Kernels and Other Kernel Architectures, pages 187-204.
- [Youn89] YOUNG, M. W. (November 1989) *Exporting a User Interface to Memory Management from a Communication-Oriented Operating System*. Technical Report CMU-CS-89-202, PhD Thesis.
- [Zajc93] ZAJCEW, R., ROY, P., BLACK, D., PEAK, C., GUEDES, P., KEMP, B., LOVERSO, J., LEIBENSBERGER, M., BARNETT, M., RABII, F., and NETTERWALA, D. (January 1993) *An OSF/1 UNIX for Massively Parallel Multicomputers*. Winter USENIX Conference, pages 37-55.
- [Zaya87] ZAYAS, E. (November 1987) *Attacking the Process Migration Bottleneck*. Proceedings of the 11th Symposium on Operating Systems Principles, pages 13-24.

## A Measurements

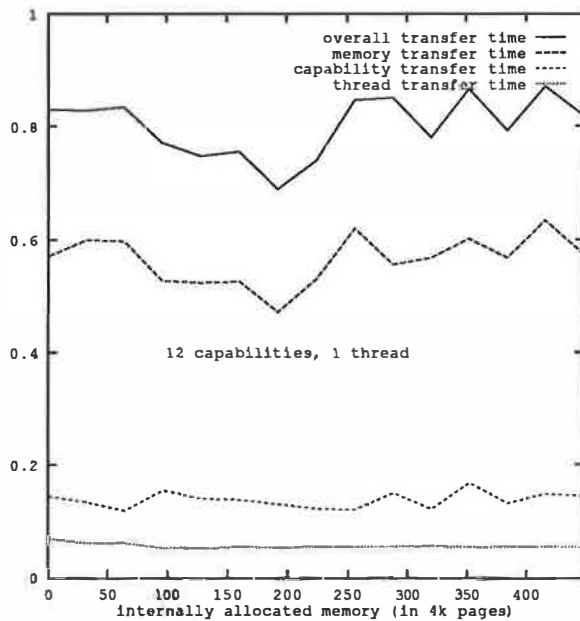


Figure 3. Transfer Time (in sec) v. Memory Size

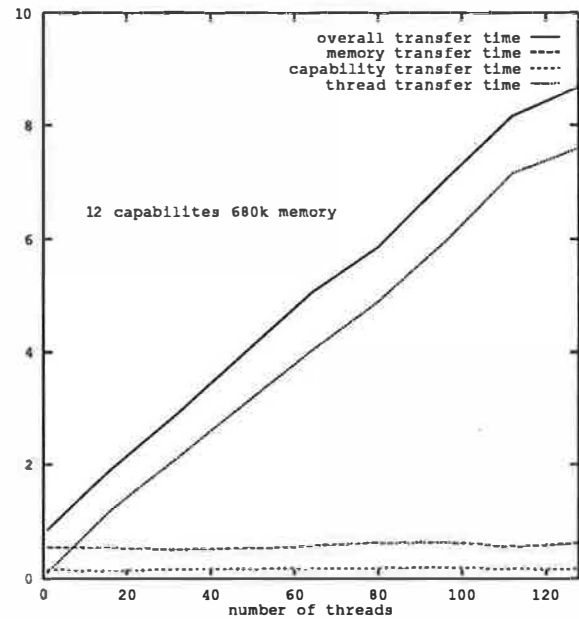


Figure 4. Transfer Time (in sec) v. Thread Number

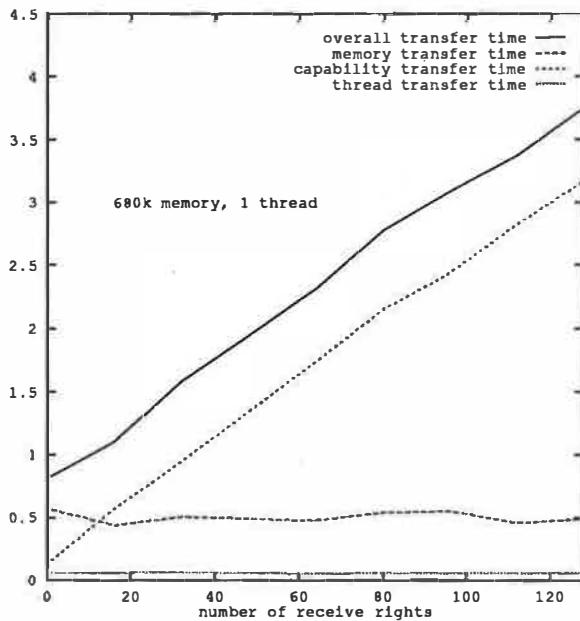


Figure 5. Transfer Time (in sec) v. Number of Receive Rights

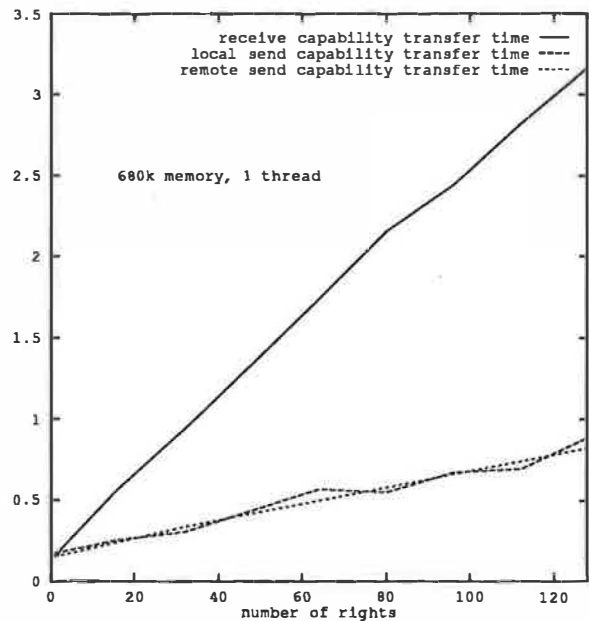


Figure 6. Rights Transfer Time (in sec) v. Number of Rights

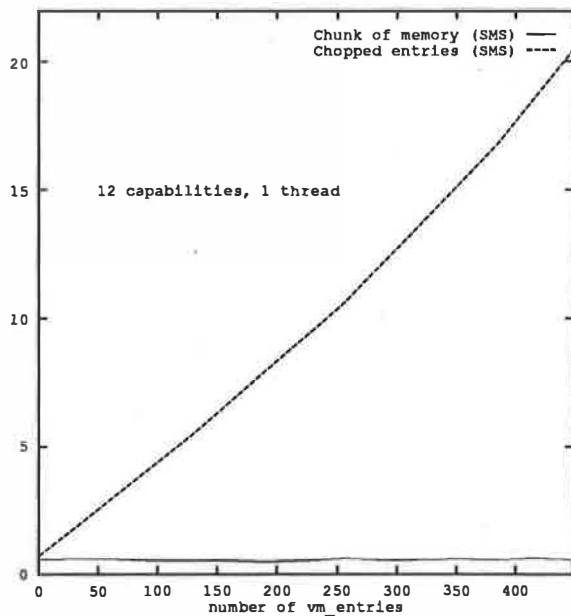


Figure 7. Transfer Time (in sec) v. Memory Contiguity

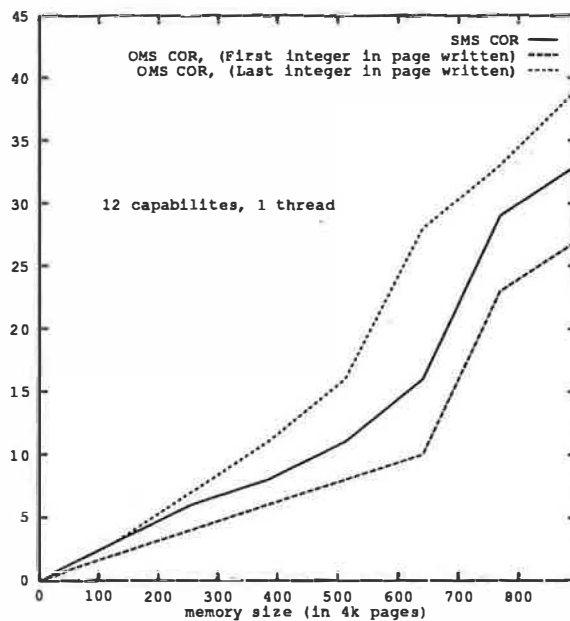


Figure 8. Execution Time (in sec) v. Memory Size, (First/Last Integer in Page Written)

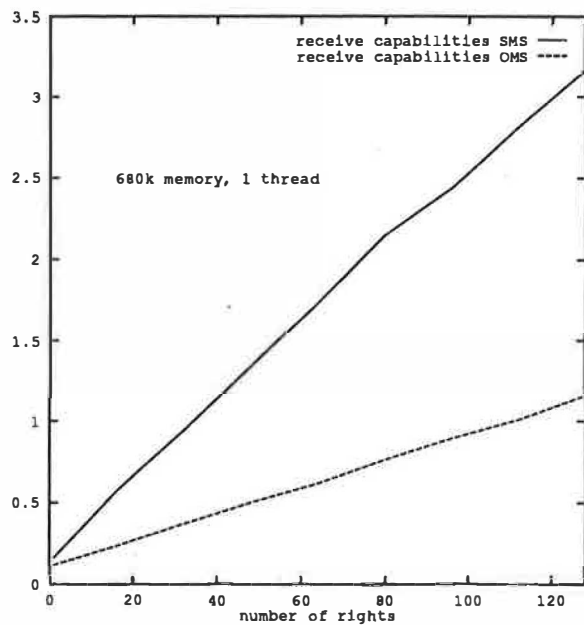


Figure 9. Transfer Time (in sec) v. Number of Receive Rights

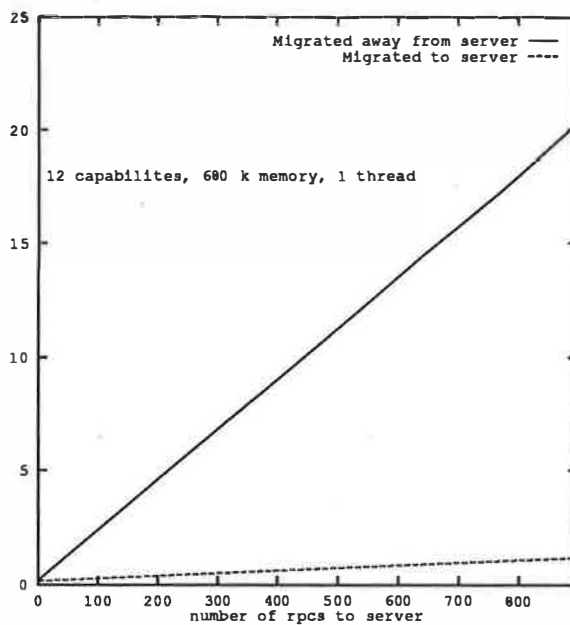
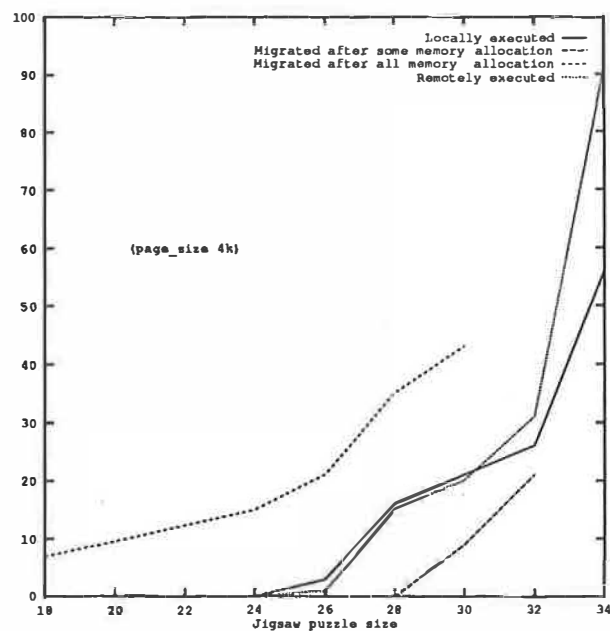
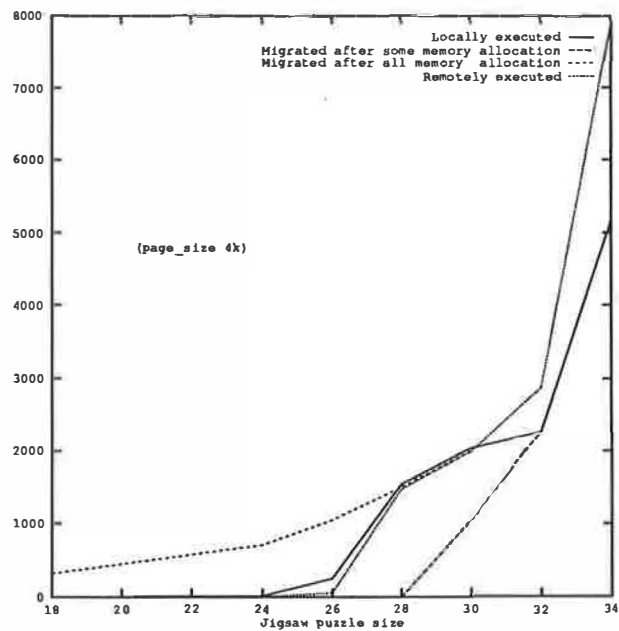


Figure 10. Execution Time (in sec) v. Remote/Local IPC



**Figure 11. Solving Time (in sec) v. Jigsaw Puzzle Size**



**Figure 12. Paging (Number of Pageins & Pageouts) v. Jigsaw Puzzle Size**

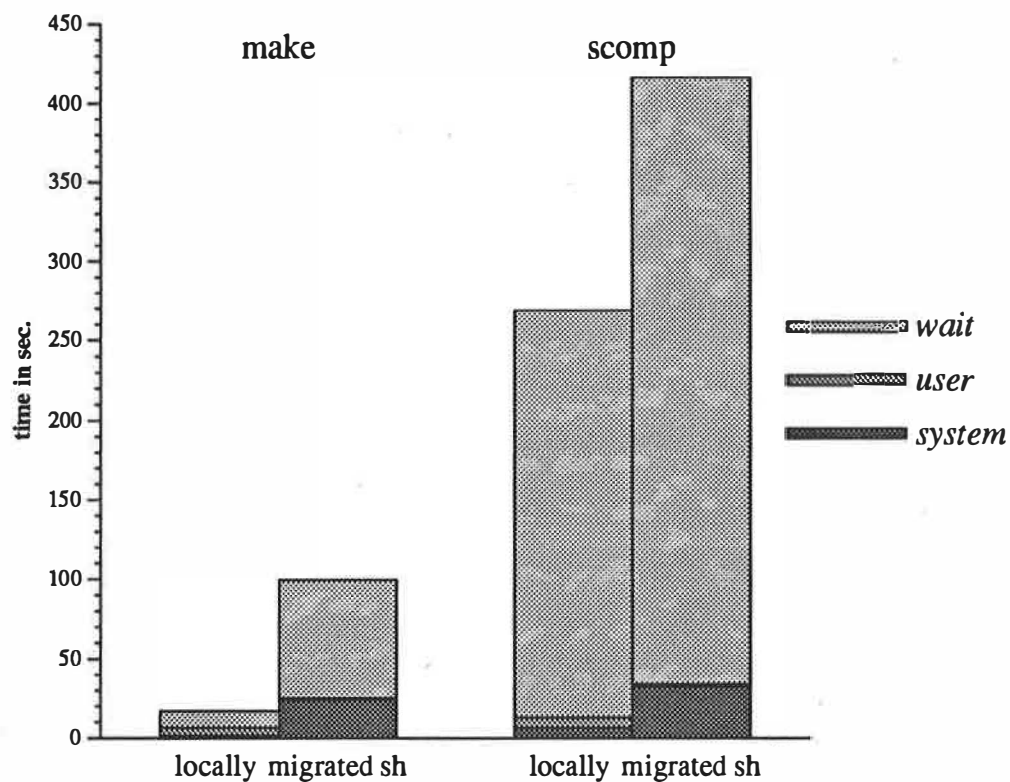


Figure 13: *make ratio*.



# The Design of the Schizophrenic Workstation System

Mark Swanson  
Terence Critchlow

Leigh Stoller  
Robert Kessler

Center for Software Science  
University of Utah  
Salt Lake City, UT 84112  
E-mail: {swanson,stoller,critchlo,kessler}@cs.utah.edu

## Abstract

*A cycle-harvesting distributed operating system is described. Building upon current kernelized operating system technology, it promises to utilize spare workstation resources for large, long-running applications, while subjecting the workstation owner/user to little inconvenience. The design is based on a novel use of the multiple-personality potential of microkernel-based operating systems. Acceptance of the system by workstation owners is encouraged by utilizing new and existing resource management mechanisms within the kernel to minimize the impact on those owners' local systems. The applications of interest may display medium-grained concurrent/parallel behavior, or they may utilize process-level parallelism, or they may simply be long-running, compute intensive, non-completion-time-critical single-threaded programs.*

## 1 Introduction and Related Work

Networks of increasingly powerful workstations are common today in many commercial and research environments. Much of the power of these workstations is wasted. Their owners spend significant portions of their time reading mail, writing, debugging and thinking; during these times, the machines, though in use, are underutilized. Much of the time the machines are essentially idle: their owners attend meetings, engage in travel, go home in the evening. Recent monitoring of workstations in the Department of Computer Science[5] clearly demonstrates underutilization. Hoogenboom measured the mean load average of 19 faculty and staff machines to be approximately .11 (five minute load averages sampled at fifteen minute intervals for two weeks). Assuming that a machine with a load of 1.0 is fully-, but not over-utilized, 89% of the power of these machines is unused. Recently those 5 to 10 MIP machines used in the study were replaced by machines rated at 76 MIPS / 23.7 MFLOPS. Although utilization has most likely fallen, even using the conservative 89% figure, some 1285 MIPS / 400 MFLOPS are available.

In contrast to this generally low average utilization, at other times the workstation owner has computational demands that consume the entire power of his workstation and quite possibly exceed its capacity. This computationally intense usage may even interfere with the workstation's use for previously mentioned, less demanding tasks. Load sharing presents a way for the workstation owner to mitigate his own impact on his workstation's interactive performance by shedding load to other, less used workstations.

### 1.1 Load Sharing

Load sharing is an area of research that has sought ways to use networks of workstations more efficiently; to shift load from overburdened workstations to less utilized ones. Load sharing systems

employ local and/or global scheduling algorithms [6, 27, 23] to statically determine which workstation is the best candidate for a new task, or when a newly arriving task should be transferred elsewhere. For example, the **Load Information Manager** in Utopia periodically gathers statistics about the participating workstations, using CPU average, memory availability and the number of users logged in to determine which workstation is the best candidate for a new task. Similar to load sharing systems are load balancing systems [19, 20, 4], which consider the system as a whole, attempting to remove imbalances in the system either by transferring new tasks as they arrive, or by migrating already running tasks. The goals of load sharing are similar to those of a cycle-harvesting system; we make the distinction that a cycle-harvesting system limits its load sharing to the use of otherwise unutilized resources. Some previous load sharing systems fit this definition as well. We note that load balancing systems do not fit the definition of cycle-harvesting, as their attempts to balance load can result directly in over utilization (albeit balanced) of all participating workstations.

## 1.2 Related Work

Many approaches to shared use of workstations have been explored, including runtime libraries, operating systems, programming languages and remote execution environments.

Distributed operating systems is an area of research that seeks to combine the power of individual workstations into larger, coordinated systems, achieving load sharing as a side effect. The various approaches that have been tried afford some level of transparency to the users. Location and reference transparency allow the user to program without knowledge of where an object is located, or whether the object is local or remote. The *port* abstraction of Chorus [18] and Mach [17] is an example of how transparency can be provided. An application simply sends a *message* to a port; the operating system is responsible for finding the receiver, and delivering it. Another example is the *RPC* mechanism of Clouds [25], which treats all services as *objects* to which invocations can either be local or remote. The goal of this type of distributed system is to create a large time-sharing system for running large applications (*possibly process level parallel*), or one on which medium-sized parallel programs can be run.

A different class of distributed system, *network operating systems*, seeks to combine networks of workstations by providing transparent access to remote resources, such as files and I/O devices, while retaining the separate operating system identities of the workstations. Locus [24], Utopia [27] and Condor [8] are examples of network operating systems that use remote access mechanisms to allow access to possibly non-local resources. In Locus, for example, remote files are accessed by converting system calls into kernel-to-kernel RPC. In Condor, system calls are handled by a *shadow* program that converts system calls into RPC. This technique requires that Condor programs be linked against a special runtime library to handle the conversion. In Utopia, a software layer implemented entirely at the user level is interposed between the application and the operating system. RPC mechanisms are used to provide global scheduling information, and remote execution.

Library-based systems such as PVM[21] and remote execution environments such as Butler[12] are another approach to load sharing. They provide the potential for wide applicability, limited by the portability of the library or environment rather than an entire operating system. For similar reasons, they are relatively simpler to implement. Their weaknesses arise from the same base as their strengths. By limiting themselves to the abstractions presented by their host operating systems, they limit their functionality and sacrifice efficiency. They often cannot properly support all direct system calls, requiring that only calls mediated by the library be used. By definition, such systems require relinking applications with the library; existing binaries simply cannot be used.

An important characteristic of a given distributed system is whether the individual workstations retain their identity. Instead of being part of a processor pool, as in Amoeba [11], workstations remain autonomous when participating in a strictly load sharing system. Systems such as Sprite [3], Condor [8] and Butler [12] are examples of this type.

## 1.3 Issues in Load Sharing

In all of these approaches common difficulties arise; some of them are technical in nature, others are sociological in nature, and some fall into a gray area in between.



The technical problems relate to how well the goal of exploiting the underutilized workstations is accomplished. The questions here relate primarily to efficiency. Efficiency depends on how much of the total capacity of the networked machines can be utilized and on how much overhead achieving that level of utilization imposes. The generality, or transparency, of an approach also determines efficiency, as it determines which applications can utilize other workstations. Other technical issues include platform heterogeneity, the ability to utilize workstations of differing architectures as a single load sharing system, and portability, the applicability of the solution on homogeneous systems of various architectures or different operating systems.

Sociological issues relate to the perception that a workstation is, in some sense, owned by the person whose desk it sits on, or who is its primary user. One of the many reasons why workstations are preferred over time-shared systems is that the workstation owner does not have to share the resources of his machine. This allows him to develop an internal model for the response time of his various work tasks and thus optimize his work patterns. When others utilize the workstation, the response time model could be invalidated and result in a significant impact on the owner's work. This would likely result in the owner disallowing all external use of the machine, making it unavailable for load sharing use. Load sharing systems generally use heuristics to determine when a workstation is idle, and when a foreign process must be evicted because the owner has computing needs of his own. In Butler, eviction is accomplished by terminating the foreign process. In Sprite and Condor, the foreign process is migrated to another workstation.

Administrative and security concerns fall in the middle ground. The workstation owner's concerns go beyond responsiveness and predictability. The owner does not want the availability of his machine threatened by other users and does not want other users to gain access to his system through load sharing mechanisms who would otherwise be denied such access. Some aspects of administration and security interact with generality: dependence on the workstation's operating system for file access, for instance, might require that all workstations have identical file name spaces. Some load sharing solutions require that users wishing to take advantage of load sharing must have accounts on all machines that they might potentially utilize[12]. This can present system administration costs unless all workstations on the network are already maintained with a homogeneous group of users, which can be impractical.

## 1.4 Addressing the Issues

The Schizophrenic Workstation system (or just Schizo) is a distributed operating system that provides a single distributed memory multiprocessor suitable for both timeshared processing and distributed memory parallel computation. It differs from distributed systems composed of dedicated (non-autonomous) workstations in that the constituent workstations do retain an autonomous identity *in addition to* participating in the distributed system. As the design of the system is described, we will show how it addresses each of the issues outlined above for load sharing systems.

The remainder of this paper is organized as follows: Section 2 describes the operating system technology that Schizo is based upon. Section 3 presents the Schizo design in terms of this technology. Section 4 discusses how the Schizo model addresses the issues raised in this introduction. Section 5 addresses shortcomings in the model. Finally, Section 6 presents some measurements and Section 7, conclusions and some areas for further research.

## 2 Base Operating System Technology

The current Schizo system is based on OSF/1 Release 1.0 [15], an existing example of an *operating system server* running as a task on top of a *kernelized* OS, in this case the NORMA[1] version of the Mach 3.0 microkernel. OSF/1 is also referred to as an *operating system personality*, insofar as it implements the high level operating system facilities we commonly think of as defining an operating system, e.g., the application interface, file system and process management. Since the server, or personality, runs as a task outside the kernel, it is possible and sometimes desirable to run multiple servers (i.e. multiple personalities) on top of a single microkernel.

Given its role as a *base* for separate operating system servers, the Mach 3.0 microkernel[13] provides only a small set of abstractions:

- the task
- the thread
- virtual memory
- ports
- messages
- devices

The NORMA version of the microkernel is so called because it contains direct support for a single system image on NO Remote Memory Access multicomputers. The direct support entails changes to the microkernel such that a collection of microkernels, each running on a separate processor, present a unified port name space, a shared virtual memory system providing distributed shared memory[7], and a mechanism to specify the processor on which a new task is created. The NORMA microkernel contains additional support for determining system configuration at system startup.

Mach identifies and manipulates most resources through the use of ports. The provision of a unified port name space means that operations on ports display both referential and location transparency. Since tasks and threads are referred to in system calls by their associated port names, this transparency also applies to them.

Location and reference transparency for virtual memory relies on the unified port name space since memory *objects* are referred to via ports. It also requires the extension of the underlying paging mechanisms, since the *contents* of memory objects are referenced by virtual address. A reference to an address whose physical page is not resident on the local node is resolved via the in-kernel memory manager, resulting in a network transfer of the appropriate page.

The microkernel implements a priority scheduler for threads and provides system calls to manipulate thread priorities. The microkernel also implements ledger objects[10] to control the use of specific resources by a task or task group. Among the resources subject to ledger control are:

- number of tasks and threads;
- virtual memory usage;
- number of ports

As an operating system personality, OSF/1 presents its own set of abstractions, which are constructed on a base of Mach abstractions. For example, an OSF/1 process is comprised of a Mach task with an associated Mach thread and virtual memory objects representing its text and data. Since an OSF/1 process is contained within a Mach task, it is often the case that referring to a “task” is equivalent to referring to a “process.” For the remainder of this paper, except where noted, the terms process and task are used interchangeably.

### 3 The Schizo Design

The Schizo system design employs features of the distributed microkernel-server architecture to marshal underutilized but autonomous workstations into a distributed system. Schizo is, in particular, an application of the multiple personality feature. Each workstation runs the Mach 3.0 NORMA microkernel. Each microkernel supports two personalities<sup>1</sup>: one is the local workstation “system”; the other is the Schizo distributed system. Only one server *task* runs on each workstation, but the server on the workstation acting as the Schizo host machine, also communicates with/utilizes

<sup>1</sup>This is really the simplest case. It is quite possible that the workstation owner might be running multiple “local” operating system personalities (e.g. OSF/1 and a DOS server[16]).

the kernels of all the other workstations and thereby uses those workstations to run its processes. This use of multiple personalities, one being the distributed system, is the key concept of the Schizo design and the reason why we named this project the “Schizophrenic Workstation” – the workstations will be schizophrenic in their use as either a dedicated workstation or as a node in the distributed system.

Although the NORMA microkernel provides a single unified port name space and an integrated virtual memory system, the ports and virtual memory objects of the two personalities using a workstation are logically partitionable, i.e., the tasks, threads, ports, and virtual memory objects of the local workstation system are unknown to the Schizo system. The same holds true in the other direction. The two systems are independent of each other and only interact indirectly through their use of the same microkernel and the underlying machine.

### 3.1 Local Scheduling

For a distributed system such as Schizo, a useful distinction between local and global scheduling can be made. The Schizo system depends on the normal Mach microkernel priority scheduler to perform local scheduling. Schizo depresses the priorities of its threads relative to the “normal” priority levels at which local tasks are expected to run. Systems that use depressed cpu priority alone have been known to display undesirable paging behavior, as the paging system is unaware of these differences in priority. We have therefore modified the kernel to provide two classes of paging service; this is treated in more depth in section 4.2.

### 3.2 Device Access

Devices are another area where interaction between servers might occur. While cooperative shared control of a device by multiple tasks is possible, we do not envision the Schizo system attempting to share devices with the resident server. There are a number of reasons for this:

- security – what cannot be accessed cannot be compromised (at least directly);
- reliability – multiple device controllers would require a high level of synchronization and co-operation, which is in contradiction to the premise that the two systems are ignorant of each other;
- flexibility – we envision allowing the workstation server to be whatever the “owner” chooses to run; requiring these servers to adhere to some device sharing policy/mechanism would limit the choices available to the user

The Schizo system does not require direct device access, nor are Schizo child tasks expected to do so. This is enforced by the fact that only the operating system server can provide the necessary capability, and the fact that Schizo child tasks cannot communicate with the local server to request it. This separation avoids interference with the local system, and it ensures that no dependencies (at the operating system level) on the availability of a particular workstation are introduced. The Schizo design tolerates the dynamic dependence of a child task on the workstation it is currently utilizing but does not tolerate *system* dependence on particular workstations (an exception to this is described in the next section).

The tasks *indirectly* utilize the network device(s) when they manipulate location transparent entities such as ports and virtual memory. Such accesses can result in the microkernel performing network transactions on their behalf. These network transactions, undertaken by the microkernel on behalf of the Schizo tasks, present another situation which can impact performance of local activities. We have not yet taken measurements which might indicate whether or not they present a noticeable load. If they do, we will prioritize access to the network devices so that Schizo-related messages will be delayed in favor of local workstation activity. In the event that incoming Schizo-related messages present an unacceptable load, heuristics will need to be implemented to quench them at their source.

Tasks also may indirectly utilize local disks for virtual memory paging activity. This occurs if the pager for the Schizo memory objects pages to local disk. Such activity is essentially unavoidable, but changes to the VM system (see Section 4.2) attempt to minimize performance impacts<sup>2</sup>.

### 3.3 Static and Dynamic Resource Base

An exception to the avoidance of dependencies on a particular workstation is the Schizo "core." To simplify the design and to provide an assured resource base, a Schizo system must have at least one dedicated processor, on which the Schizo server task runs, accepting job requests. This single dedicated processor can be generalized to a collection of processors, in which case the dedicated core of the Schizo system is essentially an OSF/1 multicomputer or cluster<sup>3</sup>. On these core machines, Schizo is the only personality/server and *does have access* to (and dependence upon) devices.

The workstations used by the Schizo system (outside of its core machines) are considered to be "owned" by their primary users and not under the control of the Schizo system. It is expected that they may come and go (shutdown or crash and reboot), often without warning to the Schizo system. The Schizo system is expected to survive the disappearance of constituent workstations outside the core and to re-assimilate such systems when they reboot. Schizo tasks, however, may disappear if their host workstation disappears. In the case of controlled shutdown, the Schizo tasks will be migrated away from the workstation to some other host. The core system forms a "haven of last resort" in the event that the pool of underutilized workstations shrinks precipitously relative to Schizo's workload. An important impact of this is that when migration occurs the task must retain no residual dependencies on the workstation it is removed from. Secondly, the core system must provide sufficient resources (especially swap/paging space) to contain all of the extant tasks, ensuring their survival, although there may not be sufficient resources to ensure their progress.

### 3.4 Global Scheduling

As a distributed system, Schizo must make decisions about task placement. In the current design, this activity is handled centrally, and rather naively, within the Schizo core. The central server regularly polls the participating workstations to gather load information. When a new Unix process is to be forked, or a process (task) needs to be migrated, a location is chosen for it based on these criteria:

- a workstation already running a Schizo task will not be selected;
- a workstation with a load above a specified threshold will not be selected;
- from the remaining workstations, the one with the lowest load, below a maximum threshold, will be selected;
- if no qualifying workstation is found, the task is created on the core system.

Workstations that are currently running Schizo tasks are ignored in favor of those that are not yet doing so based on the assumption that Schizo tasks are compute intensive and will make maximum progress if they need not compete with other Schizo tasks. In the event that Schizo determines that it has insufficient resources to ensure the new task's survival in the event of a decrease of participating workstations, it will refuse to accept the task.

The major attractiveness of this approach lies in its relative simplicity. The global scheduler is currently limited in function to identifying a candidate workstation for creation of a new process or migration of an existing one. It also detects situations in which a process must be migrated because of an increase in the load on its host node which prevents it from making progress. As a centralized mechanism, the scalability of this approach may be questionable, although some previous research[23] indicates that it will be adequate for the numbers of processors we envision, in the tens or (rarely) low hundreds of machines.

<sup>2</sup>Replacing the virtual memory object's default pager with one that paged to another device, e.g., the network, could avoid all use of the local disk. This ability is not included in the current design.

<sup>3</sup>In fact, to utilize file systems on a multiple-node core system, the OSF/1 AD variant must be used, which provides a distributed file system service.

### 3.5 Design Summary

The overriding theme of the design of Schizo is to maintain separation of the local workstation owner's system from the distributed load sharing system. This separation is accomplished by implementing these two systems as separate servers running on top of the virtual machine presented by the Mach microkernel(s). Where the microkernel does not enforce adequate separation, Schizo adheres to appropriate *conventions*, such as eschewing access to the workstations devices, to enhance the separation. Existing and new mechanisms within the microkernel are used to bias the microkernel's allocation of resources in favor of the workstation owner's system.

## 4 Evaluation of the Schizo Design

How does our Schizo design address the problems of load sharing detailed in the introduction? We shall examine in turn each of the areas identified there.

### 4.1 Efficiency

In terms of efficiency on an individual workstation basis, Schizo can achieve the same level of utilization of that workstation as a good batch or background task mechanism can achieve.<sup>4</sup> Whatever processor and physical memory resources are not required by local tasks, are given to the Schizo tasks resident on the workstation.

This is in contrast to the approach of some load sharing systems[8] that, in an attempt to be unobtrusive, cease activity on a given workstation for some fixed amount of time after any "local" activity. Such a system does well in exploiting idle workstations, but fails to utilize the unused cycles of an underutilized workstation. On the other hand, Schizo's prioritized use of both processor and physical memory allow it to exploit these underutilized machines without significantly impacting the responsiveness of local activities.

The other component of utilization is generality; i.e., what are the limitations imposed on processes that Schizo distributes to the participating workstations? The only strict limitation is that the process may not perform direct device access. All operating system services are available to any Schizo process regardless of the process' actual location and will function as if the process were resident on the same processor as the Schizo server task. This is not to say that a process that performs large numbers of system calls, especially file system operations, will enjoy the same level of performance as if it were co-located with the operating system task. Additional latency will be introduced by the required network transactions. Conversely, the operating system task may be able to process system calls more quickly in a Schizo system if the user computations have been successfully distributed to other workstations, leaving more resources for the operating system task on its processor.

### 4.2 Sociology

There is potential for sociological impact on the workstation owner both when Schizo tasks are resident and when they are not. In the latter case, it is important that the observable overhead imposed by the Schizo system is small. The system calls of local tasks are still serviced by a locally resident operating system server task. We anticipate that support of location transparency within the microkernel will result in only slightly increased service times for system calls. The other cost is an occasional microkernel response to load gathering queries from the Schizo system. Schizo requires no active participation beyond these responses, from the workstation, in terms of global scheduling.

When a Schizo task is present, the sociological impact is potentially even greater. Therefore, a major goal of the design requires that the presence of Schizo processes should be unobtrusive to the workstation owner. In a standard Mach 3.0 based system, multiple operating system servers and their child tasks are generally treated equivalently by the kernel. However, Schizo's child tasks

---

<sup>4</sup>In fact, the design of an effective mechanism for unobtrusive background tasks is a by-product of this research.

*should* be treated differently than those of the workstation owner's system, in that the latter should receive preference in any competition for resources. This is accomplished via prioritization of CPU usage and the virtual memory system.

Operating system servers are free to manipulate the priorities of the threads in their child tasks. As mentioned earlier, the thread priorities for all Schizo tasks running on "owned" workstations are depressed to ensure that the workstation's tasks receive preference for cpu resources. While this ensures that Schizo tasks run only when there are no local tasks that wish to run, this does not address the potentially adverse effects of the paging behavior that results when a large Schizo task is allowed to run.

The effects of paging behavior have been addressed by extending the microkernel to recognize two classes of memory objects. Whenever the Schizo personality creates processes on a two-personality workstation, it specifies to the microkernel that the task allocated should have an attribute specifying that all memory objects it creates be of low priority. Processes of the local workstation system generally do not have this attribute. When a page must be replaced due to a memory shortage, processes with the low priority attribute are considered first when determining which page to evict. This approach to prioritized physical memory usage was pioneered (on an earlier version of Mach) in the Stealth distributed scheduler[6].

### 4.3 Security

Beyond the promise of not significantly impacting workstation performance is prevention of denial of service. Load sharing use of a workstation should not increase the vulnerability of the workstation. In particular for Schizo, its tasks running on a workstation should not be able to compromise the microkernel (an indirect threat) or the other tasks running on that workstation (a direct threat). The independence of the servers, as described above, addresses direct negative interaction between Schizo tasks and those of the local system, including the local operating system server itself. Indirect denial of service might result from exhaustion of microkernel resources, e.g., by creation of a task large enough to consume all of the available paging space.

The microkernel's ledger system<sup>5</sup> will be used to prevent this kind of resource exhaustion by limiting the total quantity of resources (e.g. ports, memory objects, paging space, etc.) that Schizo and its tasks can consume on each workstation. This use of ledgers will serve an additional function in ensuring the survival of Schizo tasks in the event that its resource base erodes seriously. As previously mentioned (Section 3.3), the core system reserves sufficient resources to reabsorb all tasks placed on other workstations. The limits provided via ledgers will serve not only to restrain use of workstation resources but also to ensure that the system will be able to find the resources necessary to migrate the task back to the core system if that becomes necessary.

## 5 Making Schizo from OSF/1

As it comes "out of the box," the combination of a Mach 3.0 NORMA microkernel and an OSF/1 server could be used to build a distributed, multiple personality system much as we have described. What distinguishes Schizo from such a rudimentary system can be summarized as follows:

- unobtrusiveness;
- security;
- robustness;
- global scheduling;
- task migration

---

<sup>5</sup>The current version of the NORMA microkernel does not implement the ledger system. However, it is part of the OSF Draft Proposed Specification[14], and is expected to be implemented in the future.

The design of Schizo has addressed several of these issues, including scheduling, unobtrusiveness, and security. Robustness is being addressed empirically, as an implementation issue. The NORMA extensions were implemented for a multicomputer environment, in which loss of the system as result of loss of a node was deemed acceptable. With each crash of a participating node, we learn more about inter-node dependencies and how to deal with them.

## 5.1 Prototype Implementation

Our initial implementation of Schizo relies on a server task running on the core system. This server registers itself with the name-server and waits to accept requests to start processes on available workstations. The requests are presented via a frontend program that is responsible for contacting the Schizo server. In a future version of the system the Schizo server's functionality could be bundled more tightly with the OS server. The current organization eases the development process, both in terms of system building and in debugging.

A Schizo program is started by contacting the Schizo server, and handing it a *program invocation* (i.e. the program name and arguments). During the creation of the new process, Schizo selects a node on which to create the underlying Mach task. Schizo then replaces the exception port of the new task with one that it holds the receive right for. This enables it to filter and react to the exceptions the task may encounter. The new task also receives an additional send right to a *fork request* port, the receive right of which is also held by the Schizo server. This enables Schizo to intercept fork system calls so that it can find a suitable node for the new task, making it relatively easy to write *process level* parallel programs. The OSF/1 emulator has been modified to use this new send right instead of the normal one for all fork operations.

In order to facilitate the migration of Schizo tasks away from a machine that is being shutdown, the microkernel has been extended to issue a new exception to all remaining tasks when it receives a shutdown message from a server (usually the workstation OS). The Schizo server receives this exception and migrates those tasks to some other node before replying to the exception, thereby allowing the microkernel/node in question to actually shutdown. The task migration mechanism used is one developed at the University of Kaiserslautern[9], which provides both eager and lazy memory object copying. When the migration is the result of a node shutdown, lazy copying is usually fatal, as the node goes down before the memory object can be faulted to the new node.

Orderly shutdowns are one situation in which task migration must occur, but another arises when the node on which a task has been placed becomes utilized by workstation tasks to the point that the Schizo task cannot make progress. In such an event, the task is moved to a new underutilized node, if one is available, or to the Schizo core if a suitable node cannot be found. Such migrations are done lazily, with pages sent across the network as they are touched. The advantage of this approach is that the overhead of migrating the memory objects is spread out over time, rather than in a large burst that is noticeable to the workstation owner.

The Schizo prototype also includes an implementation of the Stealth Prioritized Virtual Memory System that partitions memory objects into 2 classes; low priority and normal priority. The standard microkernel paging algorithm uses a set of three queues holding active, inactive and free pages[2]. The implementation of the two classes of memory objects entails providing each class with its own set of paging queues. At creation time, an object is assigned to a class based on the priority of the task requesting its creation. One set of queues receives preferential treatment and is considered to possess higher priority in use of free pages. In addition, pagein activity for lower-priority objects is not initiated if pagein for a high priority memory object is in process<sup>6</sup>. A more detailed description and performance results can be found in [22].

---

<sup>6</sup>It is not clear at this time whether this additional mechanism is necessary, since pagein activity runs in the faulting thread's context and is thus run at a depressed priority. Experiments are being conducted now to determine its effectiveness empirically.

Architecture	Fork Time	Exec Time
HP730	.02	.033
I486	.27	.29

Figure 1: Times for (remote) fork and exec system calls. Times are in seconds.

Architecture	Fault Time
HP700	6480
I486	15240

Figure 2: Time to fault a page across the network. Pagesize is 4096 bytes. Times are in microseconds.

## 6 Measurements

A series of experiments have been performed and their results will be described in this section. By means of these experiments, we hope to:

- characterize the kind of applications that are appropriate for the Schizo system;
- demonstrate the functional capabilities of the system;
- quantify the effects of remote execution and migration on the applications;
- quantify the effects of the guest applications on the workstation tasks.

After describing the experimental testbed, each of the items above will be treated.

### 6.1 Experimental Testbed Configuration

Currently, the Schizo system implementation is a prototype and has not been deployed on personal workstations. Experiments are performed using two clusters of dedicated machines. One cluster consists of 4 HP 730 workstations with 32 MB, ethernet interconnect, and local disks on SCSI disk controllers. The other cluster consists of 4 PCs with I486 DX/50 processors, 16 MB, ethernet interconnect, and local disks on SCSI controllers. All of the machines are running NMK13.16 and OSF 1.0.4 single server. To minimize external network traffic effects, the two clusters of machines coexist on a single ethernet subnet with only one other host which serves as a gateway to the outside world.

### 6.2 Characterizing Appropriate Applications

We can quantify two characteristics of appropriate applications: minimum practical size, where size is a function of compute time and program test size, and by kind and frequency of systems calls it employs.

Figure 1 displays the times for `fork` and `exec` system calls done remotely (i.e., the process calling `fork` is on the same node as the OSF/1 server, but the child is to be created on a different node).<sup>7</sup> The paging activity involved is minimal, as the child's pages are copied on demand (lazily) and it immediately `exec's` a one-page program. The `exec'ed` program simply exits. Figure 2 shows the time to fault a page of memory across the network. The test program ensures that the pages in question are already memory resident by touching them. It then forks, with the child assigned to

<sup>7</sup>Fork and exec may be expected to improve in the OSF/1 AD server, which has some provisions for distributed process management.



System Type	HP700		I486	
System Call	Local	Remote	Local	Remote
nop	100	1240	348	3170
getpid	13	13	20	20
read	812	83170	1120	116830
write	496	49900	659	50020
stat	269	1450	769	3710
sbrk	22	22	36	39
chown	273	1380	752	breaks
create	555	133430	1376	166950
signal	375	245680	912	300330

Figure 3: Remote and local system call timings on HP700 and I486 clusters. Times are in microseconds.

Test case	Local	Remote
null.c	1	8
cpio.c	4	32

Figure 4: Remote and local C compile times for an empty file and a 1400-line program on the I486. The times are in seconds.

a different node. After the fork, the pages are touched again by the child. Taken together, we can postulate that “running” a program on another node will (on the I486) cost at least .56 seconds plus approximately 15 milliseconds per page of text. This constitutes a (by no means maximal) lower bound on the compute time that a Schizo application would need to consume to be considered “practical.”

Figure 3 lists a selection of system calls and the elapsed time for each as a local or remote call. The `getpid` and `stat` calls are handled by the emulator in the calling process’ context; thus their times are essentially the same in both cases. `Nop` is an “empty” call of no arguments with no associated actions; it is used simply to measure communication and context switch time involved in a system call. `Stat` takes roughly 5 times longer in the remote case and `read` and `write` are each two orders of magnitude slower in the remote case. Coupling this with the rather large times for remote `fork` and `exec`, an application such as `make` is currently an inappropriate application for Schizo. See Figure 4 for a comparison of local and remote compilation times

One can more generally state that any program performing large amounts of reading and writing (especially small reads and writes) and making frequent use of signals (e.g., `emacs`) is an inappropriate application.

Having ruled out certain classes of applications, we turned to those that fit the description in our introduction: long-running, compute intensive, single-threaded programs.<sup>8</sup> An example of such an application is a ray tracer implemented in Scheme[26]. Once it completes an initialization phase, it computes intensively; and the length of the computation is easily adjustable by specifying the desired number of pixels. Test scenarios could be developed with short (several minute) computations and then much longer runs of the same program and test scenario were easily achieved. Finally, the program is a “real” application, rather than a synthetic benchmark. Figure 5 lists the local and remote elapsed times for a small and moderate sized sample runs of the ray tracer. We have not yet discovered an explanation for the (apparently anomalous) speedup achieved by executing the program remotely. Though relatively large, the program’s virtual memory size did not exceed the

<sup>8</sup> We had the additional qualifier “non-completion-time-critical”; this is more descriptive of the user’s expectations than of the program.

Test size in pixels	Local	Remote
1156	12.9	13.3
13000	96.8	88.5

Figure 5: Remote and local ray trace times on the I486. Times are in minutes.

available real memory and little paging occurred after the relatively short initialization phase. We have used this program in later tests described in Sections 6.4 and 6.5.

### 6.3 Demonstrating Functionality

The desired functional behavior for Schizo encompasses the following:

- the ability to run programs on participating nodes, with automatic node selection by Schizo, and without requiring any modification, recompilation, or relinking of the application;
- the containment of any child processes within Schizo's purview;
- the migration of processes as necessitated by increases in load on host machines;
- the migration of a process as necessitated by the orderly shutdown of the host machine on which it resides.

As just one example, Utah Common Lisp (UCL) was started via Schizo. It ran a simple loop<sup>9</sup>, which allocated and discarded objects on the heap, coincidentally causing periodic garbage collections. With the help of some artificial load generated on the host workstations, the UCL program moved from node to node making and collecting garbage. The ability to perform most of the functions listed above (except for handling shutdown), is implicit in the experiments outlined in the next two sections.

### 6.4 Performance Effects on the Guest Application

To prove useful, the Schizo system must ensure that programs entrusted to it make reasonable progress when resources are available within the system. The location at which these resources are available is expected to shift as workstation owners come and go and as they shift from one task to another. We wish to make a stronger statement than a simple assurance of progress; the user should be able to form a reasonable estimate of completion time for his program. This estimate would include assumptions about the availability of resources and expected "normal" time to completion for the task. We assume that the user can observe the load on the workstation base before forming his expectations, or that he has a model, perhaps from prior experience, of the load to expect.

There are other effects on runtime that are artifacts of running on a distributed system. We addressed some of these earlier, in particular system call and paging times. When migration is taken into account, it may effect runtime both directly, in terms of the immediate cost of migration and the compute time lost while migrating, and in increased paging costs due to load on the machine the task is migrated *from*. We assume that the system has selected a lightly loaded destination for the task.

Figure 6 shows the results of an experiment intended to quantify this latter cost. An artificial load program was constructed that repeatedly loops, imposing a high compute load for a given number of seconds (the "load" column in the figure) and then becoming quiescent (sleeping) for some other specified number of seconds.

<sup>9</sup>Of course, in Common Lisp, a simple loop can touch hundreds of text and data pages.

Load Period	Migration Interval	Elapsed time in minutes	Migrations
30	180	20.36	6
30	240	18.46	6
30	300	16.66	4
0	150	16.88	6
0	210	15.81	4

Figure 6: Effects of source host load on a migrated task.

The experiment is run with three nodes, one serving as the Schizo core and two as participating “workstations”. The load program is started on the two workstations, 180 degrees out of phase (i.e. viewing the loaded/idle states as a clock). The test program (the ray tracer described earlier) is started via Schizo, which places it on whichever workstation is currently idle. When the load program on that machine enters its load imposition phase, the ray tracer is migrated to the other machine. The time to completion and number of migrations for varying migration intervals is reported. As we can see, the ray tracer, which took approximately 13 minutes as a normal, local job, is significantly degraded when the node it migrates from is very busy. We attribute this to time spent fetching pages from the busy machine. As the migration interval grows, more time is available to work with the pages once they are faulted over. This conclusion is borne out by the last two cases, where no load was imposed on the source machine; the degradation is much less.<sup>10</sup>

We have not yet formed a model for the expected degradation, but these experiments will serve as a starting point for developing one.

## 6.5 Performance Effects on the Host Workstation

The performance effects of Schizo tasks on the host workstations and their tasks has already been identified as an important consideration. We attempt to mitigate these effects in a couple of ways:

1. by depressing cpu priority and constraining competition for physical memory;
2. by migrating a task to another host when the current host becomes too busy.

We shall first examine some results that demonstrate how effective the first strategy is in minimizing the impact of guest tasks. We shall then explore what, if any, lingering effects a migrated job has on the source node.

### 6.5.1 Limiting CPU and VM Consumption

An experiment was conducted in which a script of “typical” user’s activities, including editing and sleeping, was run as a foreground task. As an example of a Schizo guest task, a Cholesky factorization was run as a background task to determine its effects on the time required to complete the script of user activities.

Figure 7 displays the results. A baseline time was established corresponding to a private workstation with no Schizo tasks. Then a “worst case” time was obtained by running the background task with no controls, i.e., as another foreground task. An appreciable degradation occurs in this case. Next the background task is run with its cpu priority depressed. The degradation experienced by the foreground task is lessened, but it is still significant. Then the background task was run with normal cpu priority and depressed VM priority. This approach led to even better results, from the point of view of the foreground task, than depressing cpu priority. Finally, depressing both cpu and VM priority led to a degradation that was barely perceptible.

<sup>10</sup>Since no load was imposed on the source machine, the migrations were artificially induced by signals to the Schizo server.

Background Activity	Foreground task Elapsed time in seconds	% Increase in Time
None	133	—
Cholesky, no control	151	13.5
Cholesky, depressed cpu	145	9.0
Cholesky, depressed VM	141	6.0
Cholesky, depressed cpu and VM	136	2.2

Figure 7: Effects of background task on foreground activity.

	Counter value after 30 seconds elapsed time	% of baseline
Baseline (no Schizo tasks)	245624169	—
Schizo task faulting pages	162747867	66.2

Figure 8: Amount of “work” performed in 30 seconds.

### 6.5.2 Residual Effects

We saw in Section 6.4 that the lazy faulting of pages that occurs after a migration has a noticeable effect on the progress of the migrated job. We modified the experiment slightly by adding a counter in the busy loop of the artificial load program, as a measure of the “work” it performed in the given number of seconds. Figure 8 shows the amount of work performed by the load program in an idle Schizo system (no migrating tasks) and the amount performed immediately after a migration occurs. The impact of the migration on the workstation is significant. It is also intuitively reasonable. Page faults, including those caused by the migrated process, are serviced by kernel threads, which run at a higher priority than user threads. There may also be a second order effect if the pages for the migrated task have been paged out to disk and need to be paged in to satisfy the faults. We have not attempted to isolate these effects, if present. These results lend support to our intuition that some additional work on depressing Schizo activity will be required to meet our goal of non-intrusiveness.

## 7 Conclusions and Further Research

The prototype that we have constructed has demonstrated that Schizo can distribute load to participating workstations without seriously impairing their performance as autonomous systems. It can automatically shift load as a consequence of changes in resource availability, and it can “rescue” processes from workstations undergoing planned shutdowns.

Other goals of the design have not yet been realized in the prototype, particularly resource control by means of ledgers. Deployment on users’ desks awaits further work on robustness and on performance of the NORMA microkernel base.

We have identified several areas in which further research is warranted. One is the addition of a checkpointing mechanism to protect Schizo processes against network and node failure. Currently, Schizo processes that are running on a node that crashes, are aborted and all local state is lost. For long running application programs, this approach is unreasonable. Users of Schizo should be confident that a long running job submitted to the system will survive node failure, continuing to run as long as the core system remains alive.

The ability to handle heterogeneous nodes is another fruitful area of research. A first approach would be to allow the memory of nodes to be shared between different architectures. Since a page is ultimately just a collection of bits, paging to a node of a different type should be feasible and

might allow applications that were paging to disk to page to other nodes. The next step of allowing forking and migration of tasks across different architectures will require significant operating system and also compiler work (for example, at well defined points, it is possible that a task could snapshot its state and transform that data to another architecture and restart).

Another area we wish to explore is extending the Schizo model of one distributed system and many autonomous workstations to one where all the workstations host a set of “overlapping” distributed systems. Rather than a single core system where jobs are submitted, jobs may be submitted on any of the participating nodes. The advantage of this approach is that there is no longer a dependency on a core system; any user on any participating node can submit a job to Schizo. The disadvantage is that the scheduling and migration facilities are now decentralized, and thus are potentially more complex in construction.

## References

- [1] J. S. Barrera. A fast Mach network IPC implementation. In *Proceedings of the Second Usenix Mach Symposium*, pages 1–12, 1991.
- [2] D. Black, J. Carter, R. MacDonald, J. Van Sciver, P. Wang, S. Mangalat, and E. Sheinbrood. OSF/1 virtual memory improvements. In *Proceedings of the Second Usenix Mach Symposium*, pages 87–104, 1991.
- [3] F. Douglass and J. Ousterhout. Transparent process migration: Design alternatives and the Sprite implementation. *Software-Practice and Experience*, 21(8):757–785, August 1991.
- [4] C. Gao, J.W.S. Liu, and M. Railey. Load balancing algorithms in homogeneous distributed systems. In *Proceedings of the International Conference on Parallel Processing*, pages 302–306, 1984.
- [5] P. Hoogenboom. System performance advisor: An expert system for Unix system performance management. Master’s thesis, Dept. Computer Science, University of Utah, Salt Lake City, Utah, 1992.
- [6] P. Krueger and Chawla R. The Stealth distributed scheduler. In *Proceedings of the 11th International Conference on Distributed Computing Systems*, pages 336–343, May 1991.
- [7] K. Li. *Shared Virtual Memory on Loosely Coupled Multiprocessors*. PhD thesis, Yale University, Sep 1986.
- [8] M. Litzkow, M. Livny, and M. Mutka. Experience with the Condor distributed batch system. In *Proceedings of the IEEE Workshop on Experimental Distributed Systems*, Oct 1990.
- [9] D. Milojicic, W. Zint, A. Dangel, and P. Giese. Task migration on top of the Mach microkernel. In *Proceedings of the Third Usenix Mach Symposium*, 1993. Submitted.
- [10] D. W. Mitchell. Mach resource control in OSF/1. In *Proceedings of the Second Usenix Mach Symposium*, pages 123–130, 1991.
- [11] S.J. Mullender, G. Rossum, A.S. Tanenbaum, R. Renesse, and H. van Staveren. Amoeba - a distributed operating system for the 1990s. *IEEE Computer Magazine*, pages 44–52, May 1990.
- [12] D. A. Nichols. Using idle workstations in a shared computing environment. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, pages 5–12, 1987.
- [13] Open Systems Foundation and Carnegie Mellon University., Cambridge MA. *MACH 3 Kernel Interface*, Jan 1992.
- [14] Open Systems Foundation and Carnegie Mellon University., Cambridge MA. *MACH 3 Kernel Interface: Draft Proposed Specification*, July 1992.

- [15] *Guide to OSF/1: A Technical Synopsis*. O'Reilly & Associates, Inc., 1991.
- [16] R. Rashid, G. Malan, D. Golub, and R. Baron. DOS as a Mach 3.0 application. In *Proceedings of the Second Usenix Mach Symposium*, pages 27–40, 1991.
- [17] R. F. Rashid. Threads of a new system. (mach, a multiprocessor operating system). *UNIX Review*, 4(8):37–49, August 1986.
- [18] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, S. Langlois, P. Leonard, and W. Neuhauser. Overview of the CHORUS distributed operating systems. Tech Report 90-25, Chorus Systemes, April 1990.
- [19] M. Schaar, K. Efe, L. Delcambre, and L. N. Bhuyan. Load balancing with network cooperation. In *Proceedings of the 11th International Conference on Distributed Computing Systems*. IEEE, May 1991.
- [20] N.G. Shivaratri and P. Krueger. Two adaptive location policies for global scheduling algorithms. In *Proceedings of the 10th International Conference on Distributed Computing Systems*, pages 502–509, 1990.
- [21] V. S. Sunderam. PVM: A framework for parallel distributed computing. In *Concurrency: Practice & Experience*, pages 315–339, Dec 1990.
- [22] M. Swanson and T. Critchlow. A background task facility for MACH-based systems. Technical report, University of Utah Computer Science Department, Apr 1993.
- [23] M. Theimer and K. Lantz. Finding idle machines in a workstation-based distributed system. In *Proceedings of the Eighth International Conference on Distributed Computing Systems*, pages 112–122, Jun 1988.
- [24] B. Walker, G. Popek, R. English, C. Kline, and G. Thiel. The LOCUS Distributed Operating System. In *Proceedings of the Ninth ACM Symposium on Operating Systems Principles*, pages 49–70, October 1983.
- [25] C. J. Wilkenloh and U. Ramachandran. The Clouds experience: Building an object-based distributed operating system. In *USENIX, Distributed and Multiprocessor Workshop*, pages 333–347, October 1989.
- [26] Benny Yih. Ray tracing in concurrent scheme. Technical Report CSS Opnote 91-02, Center for Software Sciences, Department of Computer Science, University of Utah, Salt Lake City, Utah 84112, 1991.
- [27] S. Zhou, J. Wang, X. Zheng, and P. Delisle. UTOPIA: A load sharing facility for large heterogeneous distributed computer systems. Technical Report CSRI-257, Computer Systems Research Institute, University of Toronto, April 1992.

# Sprite on Mach

Michael D. Kupfer

University of California, Berkeley†

**Sprite is a distributed operating system that supports a fast, single-image network file system and transparent process migration. Over a period of 19 months we ported Sprite to run as a server on top of the Mach 3.0 microkernel. Although the resulting server does not implement some Sprite features, it can run in an existing Sprite cluster, and it supports standard UNIX programs like vi, gcc, and make.**

**Porting Sprite to Mach was generally straightforward, though there were some difficulties. Many of the problems were related to asynchronous interactions between the Sprite server, Mach, and Sprite user processes. Others resulted from trying to maintain native Sprite's internal interfaces in the Sprite server.**

**The Sprite server is 22% smaller than an equivalent Sprite kernel, and it contains almost no machine-dependent code. These improvements should significantly simplify porting Sprite to new hardware platforms. Unfortunately, the Sprite server runs the Andrew benchmark at only 38% of the speed of native Sprite. None of the performance problems appears insurmountable, but they could require a long time to track down and fix.**

## 1. Introduction

Sprite is a distributed operating system that was developed at the University of California, Berkeley [14]. It features a fast, single-image network file system [19], transparent process migration [6], and a high-performance log-structured local file system [17]. Sprite was originally written from scratch to support the SPUR multiprocessor project at Berkeley [10], so the kernel is multi-threaded and uses fine-grain locking.

We have ported the Sprite kernel to run as a server on top of Mach 3.0. The server does not have complete Sprite functionality, but it can run most UNIX commands as a client of native Sprite file servers. We used the modified Andrew benchmark [15] to partially tune the server and to analyze the remaining performance problems.

Section 2 of this paper explains why we ported Sprite to Mach. Section 3 sketches the design of the Sprite server and discusses a few of the problems that arose during design and testing. Section 4 shows how the Sprite server is smaller and more portable than native Sprite. Section 5 shows that the server is significantly slower than native Sprite, and it explains some of the known bottlenecks. Section 6 evaluates the Sprite server, and Section 7 lists possible future work. Section 8 closes with some general conclusions from the project.

---

† Author's current address: SunSoft, Inc., 2550 Garcia Avenue, MS MTV05-40, Mountain View, CA 94043-1100, kupfer@eng.sun.com

## 2. Why mix Sprite and Mach?

The Sprite project became interested in Mach for three reasons: to make Sprite more portable, to make Sprite easier to distribute to external sites, and to verify whether Mach is a suitable platform for building distributed systems.

The Sprite group at Berkeley has always been small (5–8 people), and very few sites outside Berkeley run Sprite. Time spent doing ports, or time spent preparing and supporting external releases, is time that cannot be spent doing research, and a small group cannot easily afford this lost time. We hoped that by implementing Sprite as a Mach-based server, we could rely on the Mach community to write most of the code to support new hardware (e.g., device drivers and low-level memory management code). We also hoped that by distributing Sprite as a server, rather than as a complete operating system, we could reduce the amount of time that our staff spent supporting external sites.

On the other hand, we were concerned about the potential performance loss of a server-based system. Although the UNIX server [9] had demonstrated that a server-based UNIX system could perform adequately, and the shared memory server [8] had demonstrated that Mach was sufficiently flexible to support a system like Sprite, there were no servers that provided complete operating system support with heavy reliance on the network. We concluded that Sprite would be an excellent test case for Mach's ability to support high-functionality distributed systems.

## 3. Server design and issues

Four design goals for the Sprite server emerged from the project goals listed in Section 2.

1. The server should have as little machine-dependent code as possible, so as to make it more portable.
2. The design should be simple, so as not to complicate future Sprite research.
3. The design should minimize changes to existing Sprite code, so as to reduce development time and to simplify the eventual integration of new features and bug fixes from native Sprite.
4. Performance should be comparable to native Sprite, though slight performance degradation is acceptable in return for improved portability.

These goals led to a design that is generally similar to that of the UNIX server. Nonetheless, there are important differences between the two servers. Furthermore, there were important design issues that our reading of the Mach literature had not prepared us for.

### 3.1 Design overview

As a first approximation, Mach can be thought of as a high-level abstract machine. It provides processes, scheduling, a simple interface for accessing a process's memory, and a machine-independent interface for accessing local devices such as disks or networks. The C Threads library [5] provides threads, locks, and condition variables. Thus the first step in porting Sprite was to replace low-level native code from the Sprite kernel with equivalent Mach-based code. The second step was to replace system call stubs in Sprite's C runtime library with emulation routines that make Matchmaker RPC requests to the Sprite server. These steps are similar to the transformation that was used to port UNIX to Mach [3, 9].



The result is the system shown in Figure 1. User processes communicate with the Sprite server using Matchmaker RPCs, which use Mach interprocess communication (IPC) as a transport. The emulation code in each process can also make direct kernel requests (e.g., to allocate scratch memory). The Sprite server manipulates user processes and performs I/O by sending requests to the kernel. The server also acts as an external pager [20], so that Sprite file servers can provide backing store for processes.

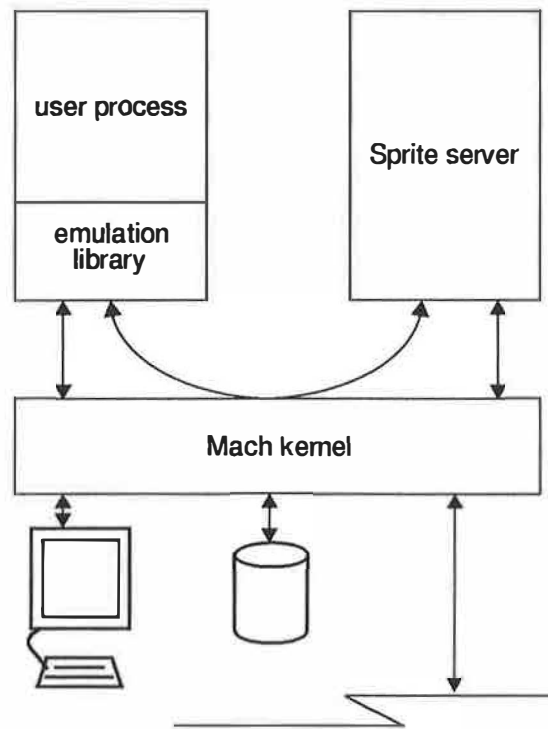


FIGURE 1. Architecture of Sprite on Mach. Each user process communicates with the Sprite server via emulation routines that use Matchmaker RPCs instead of system calls. The emulation routines and the server can also make direct kernel requests. The kernel mediates access to devices like terminals, disks, and the network.

For some code, particularly for device management, the native Sprite routines mapped well to Mach primitives, so they could be replaced by a simple facade. A few routines, such as the process scheduler and the kernel debugger stub, did not require any replacement code at all. Other code did require substantial rewriting, though as Section 4 shows, this was a small fraction of the total code. The rewritten code includes

- the guts of `fork` and `exec`, which required substantial changes to work with the Mach task and thread primitives.
- the “system call” code, which had to accept Matchmaker RPC requests instead of traps.
- an internal lock package, which was rewritten to sit on top of C Threads.
- the virtual memory code, which was rewritten to use Mach primitives and to provide a Sprite external pager.

With two exceptions, the Sprite server required no changes to the standard Mach 3.0 distribution. (This doesn't count a few bug fixes which have since been incorporated into the Mach release.) The first exception was a couple small changes to improve networking performance; see Section 5.1 for details. The second exception was to adopt the thread-local data package from the multi-server [11] version of C Threads. This package made it easier for the Sprite server to bind an internal thread to a specific Sprite user process while handling a request from that process (see Section 3.4).

The Sprite server and emulation code use Mach IPC only for local communication. For remote communication, the Sprite server uses Sprite RPCs; Mach serves only as a high-level interface to the network. There were two reasons for this approach. First, it required no changes to native Sprite servers. Second, we believed that using Mach IPC for remote communication would be slow because of performance problems with the `netmsg` server.

One difference between the UNIX and Sprite servers is how the emulation code is set up in each user process. The UNIX emulation code lives at a fixed address, is inherited when a process forks, and is typically invoked by redirecting a system trap. The Sprite emulation code does not have a dedicated address space; it is simply a part of the C runtime library and is invoked by a normal procedure call. The disadvantage of this approach is that it complicates binary compatibility. The advantage is that it was easy to implement, and we could change it later (e.g., after initial tuning) without throwing away much code.

A more important difference between the UNIX and Sprite servers is how they use external pagers. The UNIX server provides an external pager for mapped files, including program text, but it uses the Mach default pager for swap storage (i.e., a process's heap and stack). The Sprite server provides an external pager that backs the entire address space of a user process, including its heap and stack. This design lets a process page from a network file server, as is done in native Sprite. Sprite uses network paging to support diskless operation—almost all Sprite workstations are diskless—and to support process migration.

Most of these changes were straightforward to design and implement. Nonetheless, some changes presented unexpected difficulties. Many of the problems were related in some way to asynchronous interactions between the Sprite server, Sprite user processes, and Mach. The rest of this section will discuss some of the problems that arose during design and testing. None of the problems was insurmountable, but each required extra time to get the details of the design and implementation right.

## 3.2 Shared data structures

Some data structures, such as memory objects and process table entries, are logically shared between the Sprite server and Mach, and special care is needed to ensure that the data structures are managed correctly.

For example, consider the Sprite internal routine `Vm_MakeAccessible`, which converts a user address range to a kernel address range and guarantees that the kernel can access the range without faulting. The native Sprite kernel uses this routine in some places instead of `copyin`<sup>1</sup> to access user memory. Figure 2 outlines the steps for a Mach-based implementation of

---

1. The actual name of the Sprite routine is `Vm_CopyIn`, but we'll generally use the names of UNIX equivalents throughout the paper.

`Vm_MakeAccessible`. Note that Step 2 requires that the Sprite server know the name port for every memory object that it has created. Unfortunately, Mach uses an asynchronous callback to tell the Sprite server what each memory object's name port is. During early development `Vm_MakeAccessible` frequently failed because it was unable to find the name port that `vm_region` had returned, which happened because Mach had not yet registered the name port with the server. The fix for this problem was to have the code in Step 2 read from the user address before looking up the name port. This forced Mach to finish initializing the memory object, which provided the name port to the server.

1. Use `vm_region` to get the name port of the memory object for the given user address.
2. Use the name port to find the memory object (e.g., using a hash table or linked list in the server).
3. Convert the user address to the corresponding memory object offset.
4. Map the memory object page into the Sprite server.

FIGURE 2. Steps for `Vm_MakeAccessible`.

For a more involved example, consider what the Sprite server must do to clean up when a process with a mapped file exits. As shown in Figure 3, Sprite represents the mapped file as a `Vm_Segment`, which contains a handle for the file as well as ports for the corresponding Mach memory object. When the process exits, the server asks the kernel to clean any dirty pages in the memory object. It then asks the kernel to delete the memory object. Eventually the kernel responds with a callback RPC, informing the server that it can null out its remaining references to the memory object and free the `Vm_Segment`.

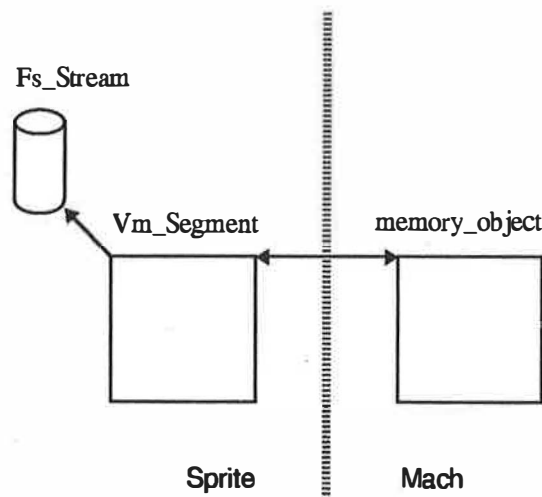


FIGURE 3. Data structures for a mapped file.

Now suppose that a second user process wants to map the same file in after the server has requested cleaning but before the server has received the callback RPC. Should the server wait until the old `Vm_Segment` is gone before creating the new one? Such an approach might lead to performance problems or deadlocks. Alternatively, should the server try to reuse the `Vm_Segment`? This would complicate code that is already somewhat hairy. Or should the server create a new `Vm_Segment`? This would invalidate an initial assumption in the server's design, which was that at most one `Vm_Segment` would exist at any time for a given file.

The Sprite server currently takes the first choice: it waits for the old `Vm_Segment` to go away before creating a new one. This approach did not seem to cause problems during development or tuning. Nonetheless, for production use it may be better to reuse the `Vm_Segment`, at least while the segment is being cleaned.

### 3.3 `copyin()` failures

Another problem that cropped up during development was how to implement `copyin` correctly. One obvious implementation of `copyin` uses `vm_read` to get the desired bytes from the user process and then uses `vm_write` or `bcopy` to write the bytes into the server's buffer<sup>2</sup>. Such an implementation is simple to code, and it can leave much of the error checking to `vm_read` (e.g., to verify that the given user address range is valid).

Unfortunately, `vm_read` only checks whether the user's address range is mapped; it doesn't check whether the bytes are actually accessible. If any page in the requested range is not in memory, the server won't discover an accessibility problem until it tries to use the bytes, the page-in fails, and the server gets an address exception. This design might be acceptable for a system such as UNIX, because the backing store is usually a local disk, and failures should be very rare. Unfortunately, this design is not acceptable for Sprite because the backing store is a Sprite file server, which is subject to arbitrary (and sometimes frequent) failures.

This means that the Sprite server must catch its own addressing exceptions. The exception handler must check whether the exception happened during a `copyin` or `copyout`, so that it can either panic or cause the copy operation to fail gracefully. The Sprite server does not currently have such an exception handler, but file server failures were infrequent enough that they did not seriously impede development or tuning. Nonetheless, it would be necessary to add an address exception handler before putting the Sprite server in production use.

### 3.4 Maintaining internal interfaces

Maintaining Sprite's internal interfaces also required some effort. Native Sprite is like UNIX in that user processes run in "kernel mode" after an interrupt or after causing a trap. As a consequence, native Sprite was designed so that a process can kill or suspend only itself. An attempt to kill or suspend a different process is simply a request, which the target process eventually acts on.

In contrast, the Sprite server is a separate Mach task that user processes invoke via RPCs. To maintain the illusion of a "current process" that has trapped into the kernel, the Sprite server uses the C Threads local data package to bind a user process to the thread that is handling that process's RPC. Unfortunately, this approach has some problems. First, special care is needed, e.g., in `exit` and `exec`, to ensure that the threads correctly manage internal resources like buffers. Second,

---

2. Of the two, `bcopy` is simpler and usually faster; see Section 5.2.2.

because the Sprite server does not handle time slice interrupts, it cannot guarantee that user processes will notice attempts to kill or suspend them. Thus, unlike in native Sprite, user processes must be able to kill or suspend other user processes. This requirement led to many changes in the locking strategy for the process management and signals code in the Sprite server, and these changes were the source of many bugs.

### 3.5 Asynchronous signal delivery

An additional problem with signals was how to invoke a user signal handler for asynchronous signals (that is, when a process invokes `kill` on some other process). The basic idea is simple: suspend the target process, push the call frame for the handler onto the stack, change the process's program counter to invoke the handler, and resume the process.

The problem is that if the process is, say, doing a Mach system call, the handler should not be invoked until the process returns from the kernel. There are ways to deal with this problem, such as

- carefully examining the process to see what it's doing
- having a dedicated thread in the user process to handle signal synchronization

but at first glance these approaches seemed like they might be slow or too complex to get right.

The Sprite server currently uses an approach similar to that of the UNIX server. Each call to the Sprite server returns a flag telling whether there is a pending signal that has a registered handler. If there is such a signal, the emulation code gets the signal information from the server and calls the handler before returning from the original call. This solution is not ideal (a signal handler can't be called until the program makes a Sprite request), but it was adequate for a prototype. In retrospect, the "dedicated thread" approach, which is how the V [4] and Spring [12] projects handled this problem, might have been the better choice.

## 4. Status and code measurements

Despite the problems mentioned in the previous section, the Sprite server works and is about 75% complete. This section explains the server's current status and presents some code size measurements.

One person (the author) developed the Sprite server over a period of 19 months. The first 7 months of the project were a halftime effort; the remaining 12 months were full-time, including 2.5 months of performance tuning. Development began on a Sun 3 but later moved to a DECStation 5000. To simplify debugging, the Sprite server runs as a UNIX application, but it only depends on UNIX for "console" access and for obtaining Mach privileged ports.

The Sprite server supports standard UNIX programs like `vi`, `gcc`, and `make`. Nonetheless, the implementation is incomplete, lacking features such as

- binary compatibility (with either the vendor operating system or native Sprite)
- local disk access
- support for debugging user processes
- process migration

As shown in Figure 4, porting Sprite to Mach let us get rid of almost all the machine-dependent code in Sprite. A Sprite DECStation kernel with the same functionality as the Sprite server contains roughly 143,000 lines of code, of which 27,000 lines are machine-dependent, including 3,600 lines of assembly code. The Sprite server contains 111,000 lines of code, of which 1,300 lines are machine-dependent. The server contains 4 lines of assembly code to help debug locking errors.<sup>3</sup>

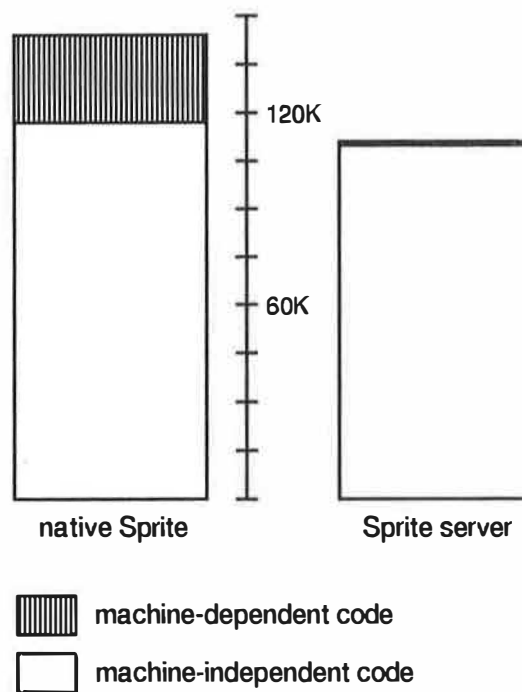


FIGURE 4. Comparison of code sizes (source lines).

Figure 5 shows how much code changed in converting the Sprite kernel to the Sprite server. We kept about 90,000 lines of code (63%). We threw away another 39,000 lines of code (27%), and we rewrote the remaining code, about 14,000 lines (10%), for use with Mach. The thrown away code consisted of low-level (and often machine-dependent) code for device, process, and memory management, plus code that duplicated Mach functionality (e.g., the process scheduler, a kernel debugger, and various C utility routines). Most of the rewritten code was for process and memory management, Matchmaker RPC processing (system calls in native Sprite), and Sprite's internal lock package.

To support the missing functionality mentioned earlier in this section, we would have to port an additional 58,000 lines of code, of which 2,400 lines are machine-dependent. About 52,000 lines of this code would probably stay the same, another 2,000 lines would probably get thrown away, and the remaining 4,000 lines would probably require rewriting.

3. The server obtains routines such as `bcopy` from the UNIX C library that is provided with Mach. Because the Sprite group would not have to maintain this code, it is not included in the Sprite server line counts.

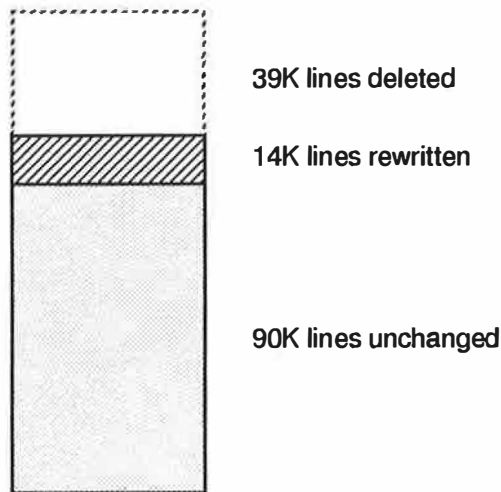


FIGURE 5. Code disposition, changing the Sprite kernel to the Sprite server.

## 5. Performance

Due to time constraints, we primarily used only one benchmark to tune the Sprite server: the modified Andrew benchmark [15]. This benchmark copies a file tree, scans the tree several times, and then compiles some window system code. It causes the Sprite server to generate around 8,200 Sprite RPCs.

This section discusses performance fixes to early versions of the server, plus current performance problems. All the performance numbers in this section were obtained on a DECStation 5000 with at least 32 megabytes of memory, using files stored on a native Sprite file server. RPC times are for two DECstation 5000s, one running native Sprite, the other running either the Sprite server or native Sprite.

### 5.1 Current status and early tuning

As shown in Figure 6, a diskless native Sprite workstation runs the Andrew benchmark in 91 seconds. The UNIX server (version UX34) and Ultrix 4.2 need around 118 seconds using files on a local disk. For an Ultrix 4.2 system accessing the files via NFS, the time goes up to 141 or 186 seconds, depending on whether the NFS server has a Prestoserve accelerator. The Sprite server's fastest time for the benchmark is 237 seconds.

Early versions of the Sprite server required 425 seconds to run the benchmark. Figure 6 shows successive improvements in the server's performance, which benefited from three general fixes: caching of text (code) segments, reduced Sprite RPC latency, and more efficient string management in `exec`.

Although native Sprite caches unused program text, we originally disabled the cache in the Sprite server so that we wouldn't have to deal with the file system and virtual memory code for detecting and removing stale text pages. Unfortunately, this meant that the server spent much of its time

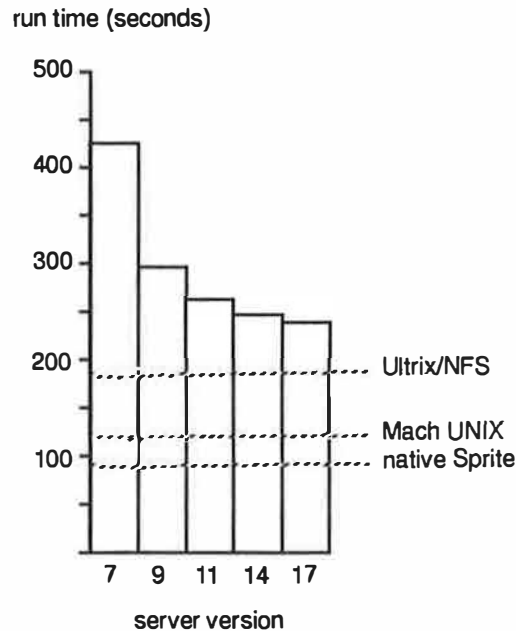


FIGURE 6. Sprite server performance. This chart shows the run time of successive versions of the Sprite server for the Andrew benchmark. Version 7 was the first version run with the benchmark. Version 9 cached unused program text. Versions 11 and 17 improved RPC performance. Version 14 used a faster version of `exec`. The chart also compares the Sprite server with related systems. Note that the Ultrix and native Sprite figures are for remote access, while the Mach UNIX figure is for local access.

faulting in the same pages for commonly used programs like `cp` and `grep`. Re-enabling the text cache reduced the benchmark run time by 130 seconds.

RPC performance in the Sprite server suffered for several reasons:

1. The UNIX server used a packet filter that accepted all packets. Thus Sprite operations would be delayed while the UNIX server read and discarded Sprite RPC packets. The solution was to fix the UNIX server's packet filter to ignore Sprite RPC packets.
2. Early versions of the server used synchronous network writes, even though they were slower than asynchronous writes. This problem was a consequence of native Sprite's interface to the network driver, and it was compounded by a server bug. Fortunately, changing the RPC code to use Mach meant that the writes could all be asynchronous, which is how the server currently operates.
3. The Sprite server was generally unable to make inband write requests because of the 128-byte size limit for inband `device` requests. This was unfortunate because inband writes were ten times faster than out-of-band writes (0.18 ms versus 1.8 ms). By raising the size limit for inband writes to 300 bytes, we were able to get the Sprite server to use inband writes for 90% of the packets that it sent.

The combination of these fixes reduced the null Sprite RPC time from 4 ms to 2.2 ms. Overall the RPC improvements (latency and throughput) translated to a 41-second speedup for the Andrew benchmark.



The other major performance fix was in the way that `exec` copied in environment and argument strings from a user process. Originally the process would pass an array of string addresses to the server, and the server would copy each string in one at a time. By changing the Matchmaker stub and Sprite emulation code to pass the strings in with the `exec` request, we were able to reduce the benchmark time by 17 seconds.

## 5.2 Current performance problems

Unfortunately, this still leaves a 146 second gap between native Sprite and the Sprite server. We can explain about 97 seconds of this gap (see Figure 7), and we have ideas for how to fix some of the problems. We know where most of the remaining 49 seconds are being spent, but not what is causing the delays. Tuning stopped in July 1992 when the Sprite server project was brought to a close.

The rest of this section will explain the current top 5 understood or partially understood bottlenecks.

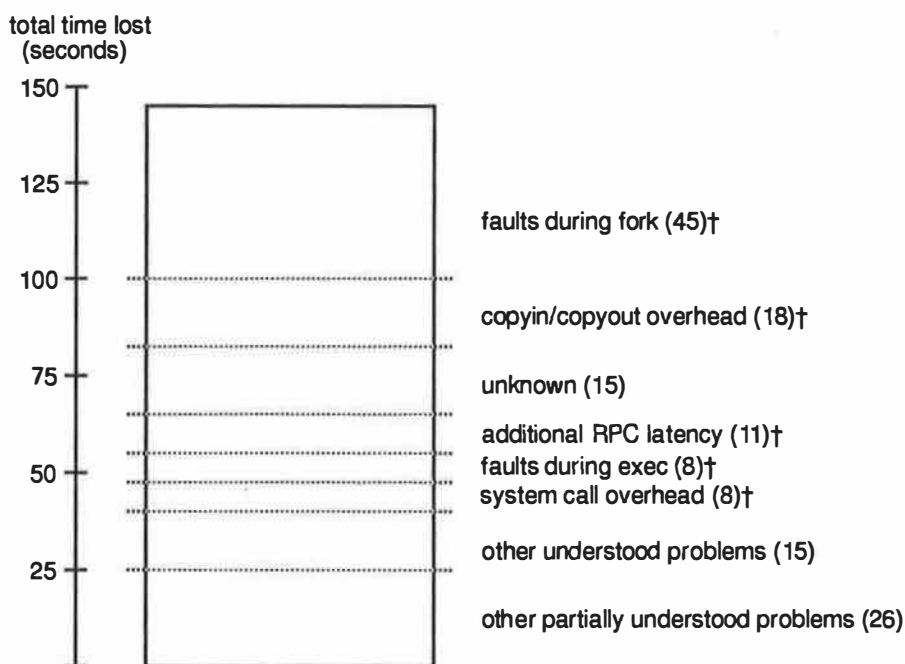


FIGURE 7. Remaining performance problems. This graph illustrates the current performance gap between the Sprite server and native Sprite for one run of the Andrew benchmark. The numbers in parentheses tell how many seconds are lost to each problem. Problems marked with a dagger (†) are discussed in the main text. "Unknown" refers to time that has not been accounted for. "Other understood problems" refers to well-understood problems that are not discussed in the paper. "Other partially understood problems" refers to problems where only the general area of the problem is known (e.g., somewhere in `fork`).

### 5.2.1 overhead in `fork()`

The largest single bottleneck is currently in `fork`, which spends an extra 45 seconds during the benchmark copying the parent's heap and stack. This copying is necessary because Mach does not currently support copy-on-write for externally managed memory objects.

The UNIX server avoids this problem by using the default pager for the heap and stack, which does support copy-on-write. It is not feasible to make the Sprite server the default pager because the default pager interface lacks the necessary hooks to support Sprite process migration. The correct solution is for Mach to support copy-on-write for externally managed objects. This support should appear as part of Joe Barrera's work to support distributed memory multiprocessors [2].

### 5.2.2 overhead in `copyin()` and `copyout()`

The second biggest bottleneck is in `copyin` and `copyout`. This bottleneck accounts for 18 seconds of the benchmark time, and it is primarily associated with `user read` and `write` calls.

The UNIX server uses mapped files to avoid this problem [9]. That is, the emulation library maps a portion of the file into the process's address space. The library satisfies `read` and `write` requests by copying directly from or into this mapped region. Unfortunately, native Sprite doesn't support consistent access to mapped files that are shared by multiple clients; it only supports consistent access using explicit `read` and `write` calls [1]. Thus we could change the Sprite server to use mapped files for `read` and `write`, but first we would have to fix native Sprite to provide consistent shared access to the files.

Fortunately, there are other options that are worth investigating. Instrumentation shows that the bottleneck results not from extra copying, but from the Mach kernel calls needed for the server to access the user process's address space. So, for example, it may suffice for the server to cache one or more mappings into the process's address space and use those mapped regions for `copyin` and `copyout`. Alternatively, it may be acceptable for the Sprite server to use mapped files in the normal (unshared) case, switching to explicit `read` and `write` calls only when the file is open for writing on multiple clients, which happens infrequently [1].

### 5.2.3 RPC latency

The third biggest bottleneck, which we estimate to be around 11 seconds for the Andrew benchmark, is higher Sprite RPC latency, despite the improvements that were made during tuning. The Sprite server requires 2.2 ms to do a null RPC to a native Sprite system, whereas a native Sprite system can do the same operation in 0.8 ms. We suspect that most of the time is spent in the network input path (packet filter, etc.), though we did not have time to verify this through measurements.

Assuming that Mach is responsible for most of the slowdown, there are two ways to fix it:

1. Improve general network performance. The work described by Reynolds and Heller [16] is an example of this approach.
2. Reduce the number of RPCs. Instrumentation shows that 36% of the Sprite RPCs that are generated by the Andrew benchmark are `open` RPCs. Adding a name cache would probably reduce this number considerably [18], though it would require changes to native Sprite.

### 5.2.4 overhead in `exec()`

The fourth biggest bottleneck is in `exec`, which appears to spend about 8 seconds in the Andrew benchmark faulting in new heap and stack pages. The problem here is that the `exec` code in the Sprite server currently destroys the heap and stack and then creates new ones. This causes an additional 8,300 external pager calls to the Sprite server, primarily to create zero-fill heap and stack pages, with each call taking an estimated millisecond of overhead. We expect that we could reclaim at least 7 of the 8 seconds by changing the server to reuse the old heap and stack.

### 5.2.5 “system call” overhead

The fifth biggest bottleneck, which accounts for another 8 seconds of benchmark time, is in the Sprite server’s “system call” code, which processes Matchmaker RPCs from user processes. The slowdown appears to come from a context switch that happens every time a new request is received. The problem is that the server thread that receives a user request is not the thread that processes it. The intent of this design was to avoid a race between getting a request and reclaiming an entry in the process control table. The correct approach, which would eliminate the context switch, would be to use no-senders notification for reclaiming process table entries.

## 6. Evaluation

This section evaluates the Sprite server according to the design goals in Section 3.

1. *Portability.* The Sprite server appears to be highly portable. As shown in Figure 4, it contains less code than an equivalent Sprite kernel, plus there is very little machine-dependent code and essentially no assembly code. The unimplemented code from native Sprite is mostly machine-independent, so a fully functional Sprite server should still be very portable.
2. *Simplicity.* Although the Sprite server is not as simple as we’d like (because of the design complications described in Section 3), most of the Sprite server design and implementation were straightforward, so we believe that the server met its simplicity goal. Furthermore, debugging the server with `gdb` was generally easy. In fact, we were able to track down bugs that were inherited from native Sprite and had long been puzzling the Sprite group.
3. *Minimize changes.* Porting Sprite to Mach involved an acceptably small number of changes to native Sprite. Most of the server code is identical to the kernel code from which it is derived, and we made few changes to Sprite’s internal interfaces. Furthermore, the Sprite server runs in an existing Sprite cluster. No changes—other than adding a new machine type—were made to the native Sprite systems to accommodate the Sprite server.
4. *Performance.* Unfortunately, the server did not meet the performance goal. Assuming that we can apply the fixes that we know of or expect to see soon, the expected performance should be about 155 seconds for the Andrew benchmark, which is only 60% of native Sprite’s performance. We believe that additional tuning could reduce the benchmark time even further, but this would probably require another 3–6 months of work.

## 7. Future work

As mentioned earlier in the paper, the Sprite server project has been discontinued. If development were to continue, the obvious first task would be to continue performance tuning. More work is needed to make the Sprite server perform adequately on the Andrew benchmark, and there are other benchmarks that the server should be tested on [7]. Furthermore, a production-quality Sprite server would require the remaining functionality mentioned in Sections 3 and 4.

Beside additional development, it would be useful to conduct research using the Sprite server. For example, it would be interesting to compare the performance of the Sprite server with the performance of a similar server that uses Mach IPC for remote device access or for process migration [13].

## 8. Conclusions

Porting Sprite to Mach was a mixed success. On the one hand, it greatly reduced the amount of machine-dependent code in Sprite, which should make Sprite much easier to port to new hardware. The asynchronous interfaces provided by Mach require some unpleasant complexity in the Sprite server, but this complexity is manageable. On the other hand, the server is almost unusably slow, even after a couple months of tuning, and it appears that much work is still needed to bring performance within striking distance of the native system.

At least one third of the performance gap results from the distributed nature of Sprite. However, the slowdown is not primarily due to RPC latency or throughput problems. Rather, it is due to the Sprite server's heavy use of an external pager, plus problems such as Sprite's inability to use mapped files to avoid copy overhead. The lesson here seems to be that there is more to high-performance distributed systems than fast communication, and although Mach shows promise as a general platform for distributed computing, it still has some serious shortcomings. For some problems (e.g., copy-on-write for external pagers) it should be possible to fix Mach, but in other cases (e.g., use of mapped files for performance), it may be necessary to redesign the distributed system instead.

If one asks whether it is worth porting an existing system to run on top of Mach, the answer seems to be "it depends." For a research system like Sprite, with its small development community, increased portability seems attractive enough to warrant a large one-time porting and tuning effort. On the other hand, a large commercial vendor, particularly a hardware vendor that would have to do its own Mach ports, would probably be better off to spend the time redesigning internal interfaces. There are other potential advantages to running on Mach, such as interoperability with different environments, but portability by itself seems inadequate justification to convert existing code to use Mach.

## 9. Availability

The sources for the Sprite server are available via anonymous ftp from `sprite.berkeley.edu` (aka `allspice.berkeley.edu`). Please note that the sources are intended only for browsing, as they may not build in a standard Mach environment, and the server will not run outside of a Sprite cluster.

## Acknowledgments

This work was supported in part by the Open Software Foundation and in part by the Defense Advanced Research Projects Agency and the National Aeronautics and Space Administration under contract NAG2-591. SunSoft, Inc. provided the time and facilities for producing this paper. The Sprite group and the Mach community both provided valuable technical assistance during the project. M. Satyanarayanan developed the original Andrew benchmark. Brent Callaghan, Dejan Milojicic, John Ousterhout, and the conference referees provided many helpful comments on

drafts of the paper. Special thanks to John Ousterhout for providing the modified benchmark and for technical assistance and guidance.

## References

- [1] Mary G. Baker, John H. Hartman, Michael D. Kupfer, Ken W. Shirriff, and John K. Ousterhout. "Measurements of a Distributed File System", *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, October 1991, 198–212.
- [2] Joseph Barrera III. private communication, September 1992.
- [3] David L. Black *et al.* "Microkernel Operating System Architecture and Mach", *Proceedings of the USENIX Microkernel Workshop*, April 1992, 11–29.
- [4] David R. Cheriton, Gregory R. Whitehead, and Edward W. Sznyter. "Binary Emulation of UNIX Using the V Kernel", *Proceedings of the Summer 1990 USENIX Conference*, June 1990, 73–86.
- [5] Eric C. Cooper and Richard P. Draves. "C Threads", Technical report CMU-CS-88-154, Carnegie Mellon University, February 1988.
- [6] F. Douglass and J. Ousterhout. "Transparent Process Migration: Design Alternatives and the Sprite Implementation", *Software—Practice & Experience* 21, 8, August 1991.
- [7] David Finkel, Robert E. Kinicki, Aju John, Bradford Nichols, and Somesh Rao. "Developing Benchmarks to Measure the Performance of the Mach Operating System", *Proceedings of the USENIX Mach Workshop*, October 1990, 83–100.
- [8] Alessandro Forin, Joseph Barrera, and Richard Sanzi. "The Shared Memory Server", *Proceedings of the Winter 1989 USENIX Conference*, January 1989, 229–244.
- [9] David Golub, Randall Dean, Alessandro Forin, and Richard Rashid. "UNIX as an Application Program", *Proceedings of the Summer 1990 USENIX Conference*, June 1990, 87–96.
- [10] M. D. Hill *et al.* "Design Decisions in SPUR", *IEEE Computer* 19, 11, November 1986, 8–22.
- [11] Daniel P. Julin, Jonathan J. Chew, J. Mark Stevenson, Paulo Guedes, Paul Neves, and Paul Roy. "Generalized Emulation Services for Mach 3.0—Overview, Experiences and Current Status", *Proceedings of the USENIX Mach Symposium*, November 1991, 13–26.
- [12] Yousef A. Khalidi and Michael N. Nelson. "An Implementation of UNIX on an Object-oriented Operating System", *Proceedings of the Winter 1993 USENIX Conference*, January 1993, 469–480.
- [13] Dejan S. Milojicic, Wolfgang Zint, Andreas Dangel, and Peter Giese. "Task Migration on Top of the Mach Microkernel", *Proceedings of the Third USENIX Mach Symposium*, April 1993.
- [14] J. Ousterhout, A. Cherenon, F. Douglass, M. Nelson, and B. Welch. "The Sprite Network Operating System", *IEEE Computer* 21, 2, February 1988, 23–36.
- [15] John K. Ousterhout. "Why Aren't Operating Systems Getting Faster As Fast as Hardware?", *Proceedings of the Summer 1990 USENIX Conference*, June 1990, 247–256.

- [16] Franklin Reynolds and Jeffrey Heller. "Kernel Support For Network Protocol Servers", *Proceedings of the USENIX Mach Symposium*, November 1991, 149-162.
- [17] Mendel Rosenblum and John K. Ousterhout. "The Design and Implementation of a Log-Structured File System", *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, October 1991, 1-15.
- [18] Ken W. Shirriff and John K. Ousterhout. "A Trace-Driven Analysis of Name and Attribute Caching in a Distributed System", *Proceedings of the Winter 1992 USENIX Conference*, January 1992, 315-331.
- [19] Brent Welch. "Naming, State Management, and User-Level Extensions in the Sprite Distributed File System", PhD thesis, University of California, Berkeley, February 1990. Also available as Technical Report UCB/CSD 90/567.
- [20] Michael Young *et al.* "The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System", *Proceedings of the 11th Symposium on Operating Systems Principles*, November 1987, 63-76.







## THE USENIX ASSOCIATION

The USENIX Association is a not-for-profit membership organization of those individuals and institutions with an interest in UNIX and UNIX-like systems and, by extension, C++, X windows, and other programming tools. It is dedicated to:

- \* sharing ideas and experience relevant to UNIX or UNIX inspired and advanced computing systems,
- \* fostering innovation and communicating both research and technological developments,
- \* providing a neutral forum for the exercise of critical thought and airing of technical issues.

Founded in 1975, USENIX is well known for its twice-a-year technical conferences, accompanied by tutorial programs and vendor displays. Also sponsored are frequent single-topic conferences and symposia. USENIX publishes proceedings of its meetings, the bi-monthly newsletter *login.*, the refereed technical quarterly, *Computing Systems*, and has expanded its publishing role in cooperation with The MIT Press with a book series on advanced computing systems. The Association actively participates in various ANSI, IEEE and ISO standards efforts with a paid representative attending selected meetings. News of standards efforts and reports of many meetings are reported in *login.*

### SAGE, the System Administrators Guild

The System Administrators Guild (SAGE) is a Special Technical Group within the USENIX Association devoted to the furtherance of the profession of system administration. SAGE brings together system administrators for professional development, for the sharing of problems and solutions, and to provide a common voice to users, management, and vendors on topics of system administration.

A number of working groups within SAGE are focusing on special topics such as conferences, local organizations, professional and technical standards, policies, system and network security, publications, and education. USENIX and SAGE will work jointly to publish technical information and sponsor conferences, tutorials, and local groups in the systems administration field.

To become a SAGE member you must be a member of USENIX as well.

There are six classes of membership in the USENIX Association, differentiated primarily by the fees paid and services provided. A description of these classes is included in this packet.

USENIX Association membership services include:

- \* Subscription to *login.*, a bi-monthly newsletter;
- \* Subscription to *Computing Systems*, a refereed technical quarterly;
- \* Discounts on various UNIX and technical publications available for purchase;
- \* Discounts on registration fees to twice-a-year technical conferences and tutorial programs and to the periodic single-topic symposia;
- \* The right to vote on matters affecting the Association, its bylaws, election of its directors and officers;
- \* The right to join Special Technical Groups such as SAGE.

For further information about membership, conferences or publications, contact:

The USENIX Association  
2560 Ninth Street, Suite 215  
Berkeley, CA 94710 USA

Email: [office@usenix.org](mailto:office@usenix.org)  
Phone: +1-510-528-8649  
Fax: +1-510-548-5738

ISBN 1-880446-49-9